DOCUMENT RESUME

ED 233 688 IR 010 778

AUTHOR	Powell, Patricia B., Ed.
TITLE	Software Validation, Verification, and Testing Technique and Tool Reference Guide. Final Report.
INSTITUTION	Boeing Computer Services, Inc., Seattle, Wash.
SPONS AGENCY	National Bureau of Standards (DOC), Washington, D.C. Inst. for Computer Sciences and Technology.
REPORT NO	NBS-SP-500-93
PUB DATE	Sep 82
CONTRACT	NB79SBCA0102
NOTE	140p.
AVAILABLE FROM	Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402 (1982-360-997/2244, $6.00).
PUB TYPE	Reference Materials - Directories/Catalogs (132)
EDRS PRICE	MF01/PC06 Plus Postage.
DESCRIPTORS	*Computer Programs; Computer Science; Glossaries; Program Descriptions; *Program Development; *Program Validation; *Selection
IDENTIFIERS	*Software Evaluation; Software Tools; *Validation Verification and Testing Techniques

ABSTRACT

Intended as an aid in the selection of software techniques and tools, this document contains three sections: (1) a suggested methodology for the selection of validation, verification, and testing (VVT) techniques and tools; (2) summary matrices by development phase usage, a table of techniques and tools with associated keywords, and an alphabetized table of keywords with associated techniques and tools; and (3) descriptions of 30 individual VVT techniques and tools. Each descriptive entry includes an accepted or invented title; a short description of the basic features of the technique or tool; a description of the input required for use; a description of the results of the technique or the output of the tool; a brief list of the actions that a user is expected to perform or an outline of method; an example to illustrate the inputs, outputs, and the method; a brief assessment of the effectiveness and usability of the technique or tool, including underlying assumptions and difficulties that can be expected in practice; an indication of the situation in which the technique or tool is likely to be useful; an estimate of the learning time and training needed to use the technique or tool successfully; a cost estimate; and a list of additional references. A 35-item glossary defines terminology used in the document. (ESR)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED233688

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

This document has been reproduced as received from the person or organization originating it.

Minor changes have been made to improve reproduction quality.

Points of view or opinions stated in this document do not necessarily represent official NIE position or policy

Computer Science and Technology

NBS Special Publication 500-93

Software Validation, Verification, and Testing Technique and Tool Reference Guide

Patricia B. Powell, Editor

Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, DC 20234

U.S. DEPARTMENT OF COMMERCE
Malcolm Baldrige, Secretary

National Bureau of Standards
Ernest Ambler, Director

Issued September 1982

IR010778

6.00

2

NATIONAL BUREAU OF STANDARDS

The National Bureau of Standards[1] was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the Nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the Nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, and the Institute for Computer Sciences and Technology.

THE NATIONAL MEASUREMENT LABORATORY provides the national system of physical and chemical and materials measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; conducts materials research leading to improved methods of measurement, standards, and data on the properties of materials needed by industry, commerce, educational institutions, and Government; provides advisory and research services to other Government agencies; develops, produces, and distributes Standard Reference Materials; and provides calibration services. The Laboratory consists of the following centers:

Absolute Physical Quantities[2] — Radiation Research — Chemical Physics — Analytical Chemistry — Materials Science

THE NATIONAL ENGINEERING LABORATORY provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

Applied Mathematics — Electronics and Electrical Engineering[2] — Manufacturing Engineering — Building Technology — Fire Research — Chemical Engineering[2]

THE INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following centers:

Programming Science and Technology — Computer Systems Engineering.

[1]Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Washington, DC 20234.
[2]Some divisions within the center are located at Boulder, CO 80303.

QC 100 .U57 NO. 500-93 1982

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

Library of Congress Catalog Card Number: 82-600589

National Bureau of Standards Special Publication 500-93
Natl. Bur. Stand. (U.S.), Spec. Publ. 500-93, 138 pages (Sept. 1982)
CODEN: XNBSAV

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1982

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
Price $6.00
(Add 25 percent for other than U.S. mailing)

TABLE OF CONTENTS

		Page
ABSTRACT and KEYWORDS		v
ACKNOWLEDGEMENTS		v
PREFACE		vi
Section 1		
1.1	Introduction	1
Section 2		
2.1	A Suggested Methodology for the Selection of V,V&T Techniques and Tools	1
2.2	Selection Aids	2
Section 3		
3.1	Selection Matrices and Keyword Tables	2
Section 4		
4.1	Introduction to Technique and Tool Descriptions	11
4.2	Algorithm analysis	12
4.3	Analytic modeling of system designs	18
4.4	Assertion generation	24
4.5	Assertion processing	31
4.6	Cause-effect graphing	35
4.7	Code auditor	38
4.8	Comparator	41
4.9	Control structure analyzer	43
4.10	Cross-reference generator	46
4.11	Data flow analyzer	50
4.12	Execution time estimator/analyzer	53
4.13	Formal reviews	55
4.14	Formal verification	60
4.15	Global round off analysis of algebraic processes	64
4.16	Inspections	68
4.17	Interactive test aids	73
4.18	Interface checker	76
4.19	Mutation analysis	79
4.20	Peer review	84
4.21	Physical units checking	93
4.22	Regression testing	96
4.23	Requirements analyzer	98
4.24	Requirements tracing	101
4.25	Software monitor	104
4.26	Specification-based functional testing	107
4.27	Symbolic execution	111
4.28	Test coverage analysis	115
4.29	Test data generators	118
4.30	Test support facilities	121

		Page
4.31	Walkthroughs	124
Glossary		129

LIST OF TABLES AND FIGURES

Table 3.1-1	Requirements Specification Selection Matrix	3
Table 3.1-2	Design Specification Selection Matrix	4
Table 3.1-3	Code Selection Matrix	5
Table 3.1-4	Alphabetized Keywords with Associated Techniques or Tool	6
Table 3.1-5	Alphabetized Techniques and Tools with Keywords	8
Table 4.1-1	Technique and Tool Description Entries	11
Table 4.3.6-1	Resource Requirements for Optimization Example	20
Table 4.3.6-2	Resource Requirements for Revised Optimization Example	21
Figure 4.2.6-1	QUICKSORT	15
Figure 4.2.6-2	MERGESORT and QUICKSORT Comparison	16
Figure 4.3.6-1	Resource Requirements for Optimization Example	20
Figure 4.3.6-2	Revised Optimization Example	22
Figure 4.4.6-1	Sort Specification	25
Figure 4.4.6-2	Sort Routine with Assertions	26
Figure 4.4.6-3	Sort Routine with an Intermediate Assertion	26
Figure 4.5.6-1	Source Program with Untranslated Assertion	30
Figure 4.5.6-2	Source Program with Translated Assertions	31
Figure 4.6.6-1	Boolean Graph	35
Figure 4.6.6-2	Decision Table	36
Figure 4.6.6-3	Test Cases	36
Figure 4.9.6-1	MIS Flow Chart	44
Figure 4.9.6-2	Goto Violation	45
Figure 4.10.6-1	Sample Cross-Reference Examples	47
Figure 4.15.6-1	Triangular Matrix Inversion	65
Figure 4.19.6-1	Subroutine Count	80
Figure 4.20.5-1	A Program Structure	87
Figure 4.23.6-1	Requirements Specification Statements	100
Figure 4.27.6-1	Symbolic Execution Example	112

ABSTRACT

Thirty techniques and tools for validation, vertification, and testing (V,V&T) are described. Each description includes the basic features of the technique or tool, the input, the output, an example, an assessment of the effectiveness and usability, applicability, an estimate of the learning time and training, an estimate of needed resources, and references.

Keywords: automated software tools; dynamic analysis; formal analysis; software testing; software verification; static analysis; test coverage; validation; V,V&T techniques; V,V&T tools.

ACKNOWLEDGMENTS

This report was funded by the National Bureau of Standards' Institute for Computer Sciences and Technology under U.S. Department of Commerce Contract NB79SBCA0102. The contributors to the report, as submitted by Boeing Computer Services Co., were Randy L. Merilatt, Mark K. Smith, and Leonard L. Tripp, assisted by Alan R. Bennett, John R. Brown, Susan C. Chew, Linda S. Hammond, William E. Howden, Leon J. Osterweil, and Richard N. Taylor. Consultation was provided by Leon G. Stucki. The views and conclusions expressed are those of the authors and do not necessarily represent the official policies of the Department of Commerce or the United States Government.

PREFACE

The following document was originally included as part of a document titled "Computer Software Validation and Verification: A General Guideline". The chapter on techniques and tools was extracted to be published as a reference manual; explanatory material was added at the beginning; reviewers' comments were incorporated into the final document. The document, being prepared under contract to the Institute for Computer Sciences and Technology, is in the public domain and is, therefore, not subject to copyright. Acknowledgement and thanks are appropriate for the following reviewers who donated their time and energy to critiquing the document:

John B. Bowen
Martha A. Branstad
Lorraine M. Duvall
Carolyn Gannon
Herbert Hecht
Raymond C. Houghton, Jr.
Sukhamay Kundu
Melba Hye-Knudsen
Frank LaMonica
David Markham
Gerald Peterson
Ed Senn
Harlan K. Seyfer
Jim Skiles
Marilyn J. Stewart
Al Sorkowitz
Susan J. Voight
Natalie C. Yopconka
Saul Zaveler

Comments pertaining to the technical content are solicited and should be directed to:

Systems and Software Technology Division
Room B266 Bldg. 225
National Bureau of Standards
Washington, D.C. 20234

1.1 Introduction

The Institute for Computer Sciences and Technology (ICST) carries out the following responsibilities under P.L. 89-306 (Brooks Act) to improve the Federal Government's management and use of ADP:

o develops Federal automatic data processing standards;

o provides agencies with scientific and technological advisory services relating to ADP;

o undertakes necessary research in computer sciences and technology.

In partial fulfillment of Brooks Act responsibilities, ICST issues Special Publications (S.P.). This document is a reference guide for techniques and tools which may be used in conjunction with a validation, verification, and testing (V,V&T) methology.

The document consists of three sections:

o A suggested methodology for the selection of V,V&T techniques and tools.

o Summary matrices by development phase usage, a table of techniques and tools with associated keywords, and an alphabetized table of keywords with associated techniques and tools.

o Description of 30 V,V&T techniques and tools.

This document can be used independently as a reference or can be used in conjunction with "Guidelines on Planning for Software Validation, Verification, and Testing" (to be published as a FIPS PUB in 1982).

A glossary, included as Appendix A, defines terminology used in this document.

2.1 A Suggested Methodology for the Selection of V,V&T Techniques and Tools

The FIPS PUB "Guidelines on Planning for Software Validation, Verification, and Testing" (to be published) explains the role of V,V&T in software development, stressing an integrated approach. V,V&T planning by identifying goals, determining factors which influence the V,V&T activity, selecting V,V&T techniques and tools, and developing a detailed V,V&T plan are explained in detail. This document is particularly helpful in the selection of techniques and tools.

Selecting techniques and tools begins with the determination of a goal - a specific, measurable outcome. For example, 90 percent statement execution is a goal. Once a goal is determined, the selection matrices (section 3) are utilized to see if a technique or tool is applicable to the selected goal. For the example above, statement coverage is checked during code execution. Referencing the code selection matrix, one finds statement coverage. Next, the alphabetized keyword table (section 3) is searched for the appropriate keyword(s). For the example, the tool for statement coverage is found to be

test coverage analyzers. The last step is to reference the technique and tool descriptions (section 4) and confirm that the technique or tool does accomplish the desired goal. For the example under test coverage analyzers, the statement "Completeness is measured in terms of the branches, statements or other elementary constructs which are used during the execution of the program over the tests", confirms that a statement coverage analyzer measures the completeness of statement execution.

2.2 Selection Aids

Tables 3.1-1,3.1-2, and 3.1-3 separate techniques and tools into the broadly defined software development phases: requirements, design, and code.

The purpose of a selection matrix is to suggest possible techniques or tools for a goal in a development phase. The goal is stated (directly or indirectly) in terms of the form or content of a development product (requirements, design, code). The matrices list V,V&T techniques and tools applicable to analyzing the form or content of a product. Specifically, manual and automated static analysis techniques and tools aid in analyzing the form of each of the three products. Dynamic and formal techniques and tools aid in analyzing the semantic content of each of the products.

Table 3.1-4 lists, alphabetically, the keywords and the associated technique or tool. It may be used to identify characteristics of the technique or tool from one of the three matrices in Tables 3.1-1, 3.1-2 or 3.1-3.

Table 3.1-5 lists each technique or tool described in section 4 with applicable keywords. It may also be used to identify the characteristics of a technique or tool.

The reader with sufficient knowledge may skip Tables 3.1-1 through 3.1-5 and go directly to the technique and tools section.

3.1 Selected Matrices and Keyword Tables

The pages that follow contain three selection matrices:

Table 3.1-1 - Requirement Specifications
Table 3.1-2 - Design Specifications
Table 3.1-3 - Code

and

Table 3.1-4 - V,V&T Techniques and Tool Keywords
Table 3.1-5 - V,V&T Techniques and Tool with Keywords

ANALYSIS TYPE	AUTOMATED TOOLS	MANUAL TECHNIQUES	REVIEWS
Static	Requirements tracing aids (Note 1) Cross-reference Data flow analyzer	Requirements tracing aids (Notes 1&2) Inspections - Selected manual application of techniques listed in column one (Note 3)	Inspections Peer review Formal reviews
Dynamic	Requirements analysis Cause-effect graphing Assertion generation Data flow analyzer	Assertion generation (Note 4) Specification-based functional testing (Note 5) Cause-effect graphing (Note 5) Walkthroughs	Walkthroughs Formal reviews
Formal	Assertion generation	Formal verification (Note 6)	

NOTES
1) The requirements indexing and cross-referencing schemes are established and documented as part of the requirements specification.
2) Requirements tracing may be performed through a totally manual process.
3) Certain techniques may be manually applied to small applications or on selected portions of a given specification. This requires planning and and preparation. The larger the amount of information being analyzed, the greater the probability of error.
4) Assertion generation is performed either for later analysis using an assertion processing tool, or for manual analysis as an adjunct to testing.
5) This is a test data generation technique/tool.
6) Axiomatic specification is necessary to support analysis.

TABLE 3.1-1
SELECTION MATRIX I REQUIREMENT SPECIFICATION

ANALYSIS TYPE	AUTOMATED TOOLS	MANUAL TECHNIQUES	REVIEWS
Static	Requirements tracing aids Cross-reference Data flow analyzer	Requirements tracing (Note 1) Inspections - Selected manual application of techniques listed in column one (Note 2)	Inspections Peer review Formal reviews
Dynamic	Cause-effect graphing	Assertion generation (Note 3) Specification-based functional testing (Note 4) Cause-effect graphing (Note 4) Walkthroughs	Walkthroughs Formal reviews
Formal	Analytic modeling of software designs (Note 6) Global roundoff analysis of algebraic processes (Note 5) Formal verification (Note 8)	Algorithm analysis Formal verification (Notes 7&8)	

NOTES
1) Requirements tracing may be performed through a totally manual process.
2) Certain techniques may be manually applied to small applications or on selected portions of a given specification. This requires planning and preparation. the larger the amount of information being analyzed, the greater the probability of error.
3) Assertion generation is performed either for later analysis using an assertion processing tool, or for manual analysis as an adjunct to testing.
4) This is a test data generation technique/tool.
5) Analyzes an algebraic algorithm, independent of a given level of specification and therefore is applicable to a design or code level specification.
6) Requires the manual development of a model, which is then run.
7) Axiomatic specification is necessary to support analysis.
8) Formal verification is a primarily manual exercise though supporting tools have been developed.

TABLE 3.1-2
SELECTION MATRIX II - DESIGN SPECIFICATIONS

ANALYSIS TYPE	AUTOMATED TOOLS	MANUAL TECHNIQUES	REVIEWS
Static	Requirements tracing Cross-reference Data flow analyzer Control structure analyzer Interface checker Physical units checking Code auditor Comparator Test data generator	Requirements tracing aids (Note 1) Inspections - Selected manual application of techniques listed in column one (Note 2)	Inspections Peer review Formal reviews
Dynamic	Assertion processing Test data generators Test support facilities Test coverage analysis Mutation analysis (Note 4) Interactive test aids Execution time estimator/analyzer(Note 5) Software monitor(Note 5) Statement coverage Symbolic evaluation	Assertion generation (Note 3) Regression testing (Note 6) Walkthroughs	Walkthroughs Formal reviews
Formal	Formal verification (Note 7)	Formal verification (Note 7)	

NOTES
1) Requirements tracing may be performed through a totally manual process.
2) Certain techniques may be manually applied to small applications or on selected portions of a given specification. This requires planning and and preparation. The larger the amount of information being analyzed, the greater the probability of error.
3) Assertion generation is performed either for later analysis using an assertion processing tool, or for manual analysis as an adjunct to testing.
4) The objective of mutation analysis is to help assess the sufficiency of the test data.
5) Assist in testing the satisfaction of performance related requirements.
6) Testing after modification of tested software, i.e., retesting.
7) Formal verification is a primarily manual exercise though supporting tools have been developed.

TABLE 3.1-3
SELECTION MATRIX III - CODE

Keywords	Technique/Tool
accuracy analysis	algorithm analysis
algorithm efficiency	algorithm analysis
amount of space (memory, disk, etc.) used	algorithm analysis
amount of work (CPU operations) done	algorithm analysis
assertion violations	assertion processing
bottlenecks	analytic modeling of software designs
boundary test cases	specification-based functional testing
branch and path identification	control structure analyzer
branch testing	test coverage analyzers
call graph	control structure analyzer
check list	inspections
code reading	peer review
completeness of test data	mutation analysis
computational upper bound, how fast	algorithm analysis
consistency in computations	physical units testing
correspondence between actual and formal parameters	interface checker
data characteristics	assertion generation
dynamic testing of assertions	assertion processing
environment simulation	test support facilities
evaluation along program paths	symbolic execution
execution monitoring	software monitors
execution sampling	software monitors
execution support	test support facilities
expected inputs, outputs, and intermediate results	assertion generation
expected versus actual results	comparator
file (or other event) sequence errors	data flow analyzer
formal specifications	assertion generation
functional interrelationships	requirements analyzer
global information flow	interface checker
go/no go decisions	formal reviews
hierarchical interrelationships of modules	control structure analyzer
information flow consistency	requirements analyzer
inspections	peer review
inter-module structure	cross-reference generators
loop invariants	assertion generation
manual simulation	walkthroughs
module invocation	control structure analyzer
numerical stability	global roundoff analysis of algebraic processes

TABLE 3.1-4
V,V&T TECHNIQUE AND TOOL KEYWORDS

Keywords	Technique/Tool
path testing	test coverage analyzers
performance analysis	requirements analyzer
physical units	assertion generation
portability analyzer	code auditor
program execution characteristics	execution time estimator/ analyzer
	software monitors
proof of correctness	formal verification
regression testing	comparator
requirements indexing	requirements tracing
requirements specification analysis	cause-effect graphing
requirements to design correlation	requirements tracing
requirements walkthrough	requirements analyzer
retesting after changes	regression testing
round-robin reviews	peer reviews
rounding error propagation	global roundoff analysis of algebraic processes
selective program execution	interactive test aids
standards checker	code auditor
statement coverage	test coverage analyzers
statement testing	test coverage analyzers
status reviews	formal reviews
system performance prediction	analytic modeling of software designs
technical review	peer review
test case preparation (definition and specification)	test data generators
test data generation	mutation analysis
	specification-based functional testing
test harness	test support facilities
testing thoroughness	test coverage analyzers
type checking	interface checker
uninitialized variables	data flow analyzer
unused variables	data flow analyzer
variable references	cross-reference generators
variable snapshots/tracing	interactive test aids
verification of algebraic computation	symbolic execution
walkthroughs	peer reviews

TABLE 3.1-4 (Continued)
V,V&T TECHNIQUE AND TOOL KEYWORDS

Technique/Tool	Keywords
Algorithm Analysis	algorithm efficiency amount of work (CPU operations) done computational upper bound, how fast amount of space (memory, disk, etc.) used accuracy analysis
Analytic Modeling of Software Designs	system performance prediction bottlenecks
Assertion Generation	formal specifications data characteristics physical units loop invariants expected inputs, outputs and intermediate results
Assertion Processing	assertion violations dynamic testing of assertions
Cause-Effect Graphing	test case design using formal specification requirements specification analysis
Code Auditor	standards checker portability analyzer
Comparator	regression testing expected versus actual results
Control Structure Analyzer	call graph hierarchical interrelationships of modules module invocation branch and path identification
Cross-Reference Generators	inter-module structure variable references
Data Flow Analyzer	uninitialized variables unused variables file (or other event) sequence errors
Execution Time Estimator/Analyzer	program execution characteristics
Formal Reviews	go/no go decisions status reviews
Formal Verification	proof of correctness

TABLE 3.1-5
V,V&T TECHNIQUE/TOOL WITH KEYWORDS

Technique/Tool	Keywords
Global Roundoff Analysis of Algebraic Processes	numerical stability rounding error propagation
Inspections	check list
Interactive Test Aids	selective program execution variable snapshots/tracing
Interface Checker	correspondence between actual and formal parameters type checking global information flow
Mutation Analysis	test data generation completeness of test data
Peer Review	technical review code reading round-robin reviews walkthroughs inspections
Physical Units Testing	consistency in computations
Regression Testing	retesting after changes
Requirements Analyzer	functional interrelationships information flow consistency performance analysis requirements walkthrough
Requirements Tracing	requirements indexing requirements to design correlation
Software Monitors	execution sampling execution monitoring program execution characteristics
Specification-based Functional Testing	test data generation boundary test cases
Symbolic Execution	evaluation along program paths verification of algebraic computation
Test Support Facilities	test harness execution support environment simulation

TABLE 3.1-5 (Continued)
V,V&T TECHNIQUE/TOOL WITH KEYWORDS

Technique/Tool	Keywords
Test Coverage Analyzers	branch testing statement testing statement coverage path testing testing thouroughness
Test Data Generators	test case preparation (definition and specification)
Walkthroughs	manual simulation

TABLE 3.1-5 (Continued)
V,V&T TECHNIQUE/TOOL WITH KEYWORDS

4.1 INTRODUCTION TO TECHNIQUE AND TOOL DESCRIPTIONS

Each technique and tool description is alphabetically presented in a standard format. The following table describes the entries for each where "n" is the section number.

4.n.1. Name
This is the accepted title, or when an appropriate one does not exist, an invented title.

4.n.2. Basic Features
A short description of the technique or tool.

4.n.3. Information Input
A description of the input required for use.

4.n.4. Information Output
A description of the results of the technique or the output of the tool.

4.n.5. Outline of Method
A brief list of the actions that a user is expected to perform.

4.n.6. Example
An example to illustrate the inputs, outputs, and the method.

4.n.7. Effectiveness
A brief assessment of the effectiveness and usability, including underlying assumptions and difficulties that can be expected in practice.

4.n.8. Applicability
An indication of the situation in which the technique is likely to be useful.

4.n.9. Learning
An estimate of the learning time and training needed to use the technique or tool successfully.

4.n.10. Cost
An estimate of the resources needed.

4.n.11. References
Sources of additional information.

TABLE 4.1-1
TECHNIQUE AND TOOL DESCRIPTION ENTRIES

4.2.1. Name. Algorithm Analysis.

4.2.2. Basic features. Two phases of algorithm analysis can be distinguished: "a priori analysis" and "a posteriori testing." In a priori analysis a function (of some relevant parameters) is devised which bounds the algorithm's use of time and space to compute an acceptable solution. The analysis assumes a model of computation such as: a Turing machine, RAM (random access machine), general purpose machine, etc. Two general kinds of problems are usually treated: (1) analysis of a particular algorithm; and (2) analysis of a class of algorithms. In a posteriori testing actual statistics are collected about the algorithm's consumption of time and space while it is executing.

4.2.3. Information input.

a. Specification of algorithm

b. Program representing the algorithm

4.2.4. Information output.

a. A priori analysis
Confidence of algorithms' validity
Upper and lower computational bounds
Prediction of space usage
Assessment of optimality

b. A posteriori testing
Performance profile

4.2.5. Outline of method.

a. A priori analysis

Algorithms are analyzed with the intention of improving them, if possible, and for choosing among several available for a problem. The following criteria may be used:

Correctness
Amount of work done
Amount of space used
Simplicity
Optimality
Accuracy analysis

Correctness. There are three major steps involved in establishing the correctness of an algorithm.

(1) Understand that an algorithm is correct if, when given a valid input, it computes for a finite amount of time and produces the right answer.

(2) Verify that the mathematical properties of the method and/or formulas used by the algorithm are correct.

(3) Verify by mathematical argument that the instructions of the algorithm do produce the right answer and do terminate.

Amount of work done. A priori analysis ignores all of the factors which are machine or programming language dependent and concentrates on determining the order of magnitude of the frequency of execution of statements. For denoting the upper bound on an algorithm, the O-notation is used. The following notational symbols are used in the following description: **=exponentiation; [ ]=subscription.

Definition, f(n) = O(g(n)) if and only if there exist two positive constants C and n[o] such that $f(n) \leq C\, g(n)$ for all $n \geq n[o]$.

The most common computing times for algorithms are: O(1)<O(log n)<O(n)<O(nlog n)<O(n**2)<O(n**3) and O(2**n), O(1) means that the number of executions of basic operations is fixed and hence the total time is bounded by a constant. The first six orders of magnitude are bounded by a polynomial. However, there is no integer such that n**m bounds 2**n. An algorithm whose computing time has this property is said to require exponential time. There are notations for lower bounds and asymptotic bounds (see reference (4) for details). The term "complexity" is the formal term for the amount of work done, measured by some complexity (or cost) measure.

In general the amount of work done by an algorithm depends on the size of input. In some cases, the number of operations may depend on the particular input. Some examples of size are:

Problem	Size of input
1. Find X in a list of names	The number of names in the list
2. Multiply two matrices	The dimensions of the matrices
3. Solve a system of linear equations	The number of equations and solution vectors

To handle the situation of the input affecting the performance of an algorithm, two approaches (average and worst-case analysis) are used. The average approach assumes a distribution of inputs and then calculates the number of operations performed for each type of input in the distribution and then computes a weighted average. The worst-case approach calculates the maximum number of basic operations performed on any input of a fixed size.

Amount of Space Used. The number of memory cells used by a program, like the number of seconds required to execute a program, depends on the particular implementation. However, some conclusions about space usage can be made by examining the algorithm. A program will require storage space for the instructions, the constants, and variables used by the

program, and the input data. It may also use some work space for manipulating the data and storing information needed to carry out its computations. The input data itself may be representable in several forms, some which require more space than others. If the input data has one natural form - for example, an array of numbers or a matrix - then we analyze the extra space used aside from the program and the input. If the amount of extra space is constant with respect to the input size, the algorithm is said to work "in place".

Simplicity. It is often, though not always, the case that the simplest and most straightforward way of solving a problem is not the most efficient. Yet simplicity in an algorithm is a desirable feature. It may make verifying the correctness of the algorithm easier, and it makes writing, debugging and modifying a program for the algorithm easier. The time needed to produce a debugged program should be considered when choosing an algorithm, but, if the program is to be used very often, its efficiency will probably be the determining factor in the choice.

Optimality. Two tasks must be carried out to determine how much work is necessary and sufficient to solve a problem.

(1) Devise what seems to be an efficient algorithm; call it A. Analyze A and find a function such that, for inputs of size n, A does at most g(n) basic operations.

(2) For some function f, prove a theorem that for any algorithm in the class under consideration there is some input of size n for which the algorithm must perform at least f(n) basic operations.

If the functions g and f are equal, then the algorithm A is optimal.

Accuracy analysis. The computational stability of an algorithm is verified by determining that the integrity of round off accuracy is maintained. It is done manually at the requirements or specification level.

b. A Posteriori Testing

Once an algorithm has been analyzed, the next step is usually the confirmation of the analysis. The confirmation process consists first of devising a program for the algorithm on a particular computer. After the program is operational, the next step is producing a "performance profile"; that is, determining the precise amounts of time and storage the program will consume. To determine time consumption, the computer clock is used. Several data sets of varying size are executed and a performance profile is developed and compared with the predicted curve.

A second way to use the computer's timing capability is to take two programs which perform the same task whose orders of magnitude are identical and compare them as they process data. The resulting times will show which, if either, program is faster. Changes to a program which do not alter the order of magnitude but which purport to speed up the program also can be tested in

this way.

4.2.6. Example. QUICKSORT is a recursive sorting algorithm (5). Roughly speaking, it rearranges the keys and splits the file into two subsections, or subfiles, such that all keys in the first section are smaller than all keys in the second section. Then QUICKSORT sorts the two subfiles recursively (i.e., by the same method), with the result that the entire file is sorted.

Let A be the array of keys and let m and n be the indices of the first and last entries, respectively, in the subfile which QUICKSORT is currently sorting. Initially, m = 1 and n = k. The PARTITION algorithm chooses a key K from the subfile and rearranges the entries, finding an integer j such that for $m \le i < j$, $A(i) \le K$; $A(j) = K$; and for $j < i \le n$, $A(i) \ge K$. K is then in its correct position and is ignored in the subsequent sorting.

QUICKSORT can be described by the following recursive algorithm:

```
QUICKSORT (A,m,n)
if m<n then do          PARTITION (A,m,n,i,j)
                                  QUICKSORT (A,m,j)
                                  QUICKSORT (A,i,n)
                                  end
```

Figure 4.2.6-1 QUICKSORT

The PARTITION routine may choose as K any key in the file between A(m) and A(n); for simplicity, let K = A(m). An efficient partitioning algorithm uses two pointers, i and j, initialized to m and n+1, respectively, and begins by copying K elsewhere so that the position A(i) is available for some other entry. The location A(i) is filled by decrementing j until $A(j) \le K$, and then copying A(j) into A(i). Now A(j) is filled by incrementing i until $A(i) \ge K$, and then copying A(i) into A(j). This procedure continues until the values of i and j meet; then K is put in the last place. Observe that PARTITION compares each key except the original in A(m) to K, so it does n-m comparisons. See (5) for further details.

Worst Case Analysis. If when PARTITION is executed A(m) is the largest key in the current subfile (that is, $A(m) \ge A(i)$ for $m \le i \le n$), then PARTITION will move it to the bottom to position A(n) and partition the file into one section with n-m entries (all but the bottom one) and one section with no entries. All that has been accomplished is moving the maximum entry to the bottom. Similarly, if the smallest entry in the file is in position A(m), PARTITION will simply separate it from the rest of the list, leaving n-m items still to be sorted. Thus if the input is arranged so that each time PARTITION is executed, A(m) is the largest (or the smallest) entry in the section being sorted, then let p = n-m+1, the number of keys in the unsorted section, then the number of comparisons done is

$$\sum_{p=2}^{k} (p-1) = \frac{k(k-1)}{2}.$$

Average Behavior Analysis. If a sorting algorithm removes at most one inversion from the permutation of the keys after each comparison, then it must do at least (n**2-n)/4 comparisons on the average. QUICKSORT, however, does not have this restriction. The PARTITION algorithm can move keys across a large section of the entire file, eliminating up to n-2 inversions at one time. QUICKSORT deserves its name because of its average behavior.

Consider a situation in which QUICKSORT works quite well. Suppose that each time PARTITION is executed, it splits the file into two roughly equal subfiles. To simplify the computation, assume that n = 2**p -1 for some p. The number of comparisons done by QUICKSORT on a file with n entries under these assumptions is described by the recurrence relation

```
R(p) = (2**p)-2+2R(p-1)
R(1) = 0
```

The first two terms in R(p), (2**p)-2, are n-1, the number of comparisons done by PARTITION the first time. The second term is the number of comparisons done by QUICKSORT to sort the two subfiles, each of which has (n-1)/2, or (2**(p-1)) -1, entries. Expand the recurrence relation to get

$$R(p) = (2^{**}p)-2+2R(p-1) = (2^{**}p)-2+2(2^{**}(p-1)-2)+4R(p-2)$$
$$= (2^{**}p)-2+(2^{**}p)-4+(2^{**}p)-8+8R(p-3)$$

thus

$$R(p) = \sum_{i=1}^{p-1} (2^{**}p)-(2^{**}i) = (p-1)(2^{**}p)- \sum_{i=1}^{p-1} 2^{**}i$$

$$= ((p-1)2^{**}p)-((2^{**}p)-2) = \log n\ (n+1)\ -n+1$$

Thus if A(m) were close to the median each time the file is split, the number of comparisons done by QUICKSORT would be of the order (nlog n). If all permutations of the input data are assumed equally likely, then QUICKSORT does approximately 2nlog n comparisons.

Space Usage. At first glance it may seem that QUICKSORT is an in-place sort. It is not. While the algorithm is working on one subfile, the beginning and ending indices (call them the borders) of all the other subfiles yet to be sorted must be saved on a stack, and the size of the stack depends on the number of sublists into which the file will be split. This, of course, depends on n. In the worst case, PARTITION may split off one entry at a time in such a way that n pairs of borders are stored on the stack. Thus, the amount of space used by the stack is proportional to n.

n	1000	2000	3000	4000	5000
MERGESORT	500	1050	1650	2250	2900
QUICKSORT	400	850	1300	1800	2300

(Time is in milliseconds)

Figure 4.2.6-2 MERGESORT and QUICKSORT Comparison

Testing. The results of comparing QUICKSORT and MERGESORT are reported in reference (4) and are summarized in figure 4.2.6-2.

4.2.7. Effectiveness. Algorithm analysis has become an important part of computer science. The only issue that limits its effectiveness is that a particular analysis depends on a particular model of computation. If the assumptions of the model are inappropriate then the analysis suffers.

4.2.8. Applicability. An analysis of an algorithm can be limited by the current state of the art and the ingenuity of the analyst.

4.2.9. Learning. Algorithm analysis requires significant training in mathematics and computer science. Generally, it will be done by a specialist.

4.2.10. Costs. The cost to analyze an algorithm is dependent on the complexity of the algorithm and the amount of understanding about algorithms of the same class.

4.2.11. References.

(1) BENTLY, J.L., "An Introduction to Algorithm Design", Computer,Feb. 1979.

(2) WEIDE, B., "A Survey of Analysis Techniques for Discrete Algorithms," Computing Surveys,Vol. 9, No. 4, Dec. 1977.

(3) AHO, A.V., HOPCROFT, J.E., and ULLMAN, J.D., "The Design and Analysis of Computer Algorithms," Addison-Wesley,Reading, Mass., 1974.

(4) HOROWITZ, E., and SAHNI, S., "Fundamentals of Computer Algorithms," Computer Science Press,Potomac, Maryland, 1978.

(5) HOARE, C.A.R., "Partition (Algorithm 63) and QUICKSORT (Algorithm 64)", Communications of the ACM,Vol.4, No.7,pp.321,July 1961.

(6) HOARE,C.A.R.,"QUICKSORT",Computer Journal,Vol.5,No.1,1963.

4.3.1. Name. Analytic Modeling of System Designs.

4.3.2. Basic features. The purpose is to provide performance evaluation and capacity planning information on a system design. The process follows the top down approach to design through hierarchical levels of resolution. It can be applied at early design stages when functional modules are relatively large and where knowledge of their execution behavior may be imprecise. As the design proceeds and the modules are further resolved, the estimates of their behavior and execution resource characterization become more precise. The approach is predicated on two representational bases: on extended execution graph models of programs and systems and on extended queueing network models of computer system hardware resources and workloads.

4.3.3. Information input. The information which is needed for this technique consists of functional design and performance specifications as follows:

a. Identification of the functional components of the software design to be modeled.

b. Identification of the execution characteristics (primarily, execution time estimate) of each functional component.

c. An execution flow graph which gives the definition of the order of execution of the various functional components.

d. Execution environment specifications which can include information such as operating system overhead and the workload on the system that could potentially impact the particular software under development.

e. System execution scenarios which provide the definitions of the external inputs to the model needed for each simulation of the model.

f. Performance goals for the total system and components (an example is an upper bound for the mean and variance of the response time for a specified execution environment and scenario).

4.3.4. Information output. Output from the technique will consist of the following:

a. A lower bound on the performance of the system.

b. A comparison of the performance goals with the performance results.

c. Identification of the functional components which had the greatest effect on system performance.

4.3.5. Outline of method. Much of the effort in using this technique comes in the preparation of the necessary input information. Once this has been done, it is generally submitted to a computer which performs the simulation of the execution of the model and reports the results, which are then analyzed and the model revised as necessary. The specific steps in the technique are

as follows:

a. The structure of the software design is characterized in terms of its functional components. In that software designs are generally hierarchical in structure, a model may be modified to represent the system at different levels of detail, each being analyzed at different stages in the process.

b. The order of execution of the components is determined and the execution graph is constructed.

c. Resource requirements (e.g., hardware or operating system resources) of the functional components are identified and a possible environment is studied with the specific resource workloads being determined. These workloads consist of the average wait and usage times for the resources controlled by the environment and used by the software (such as average disk access time).

d. The workloads are then mapped into the model (as represented by the execution graph) based upon the identified environment resource requirements of the individual functional components.

e. Next, the system execution scenarios are constructed. The external inputs comprising each scenario may be formulated, for example, in terms of the number of disk accesses required to find a needed data item within a particular component.

f. Upon completion of the above steps, the model is driven, producing system and component performance results. (The "driving" of the model is usually done using a system simulation tool such as GPSS, General Purpose Systems Simulator, on a coded specification of the model.)

g. The performance results are now compared with the performance goals of the system. If the goals are not met, performance critical components are then analyzed in order to determine where improvements can be made. The design is modified and the technique repeated. This process continues until the performance is acceptable or until it can be determined that the goals are unreasonable.

4.3.6. Example. Finite element analysis is a technique for determining characteristics such as deflections and stresses in a structure (i.e., building, airplane, etc.) otherwise too complex for closed form mathematical analysis. The structure is broken into a network of simple elements (beams, shells, or cubes depending on the geometry of the structure), each of which has stress and deflection characteristics defined by classical theory.

Determining the behavior of the entire structure then becomes a task of solving the resulting set of simultaneous equations for all elements.

The example developed below is a portion of a system which does a finite element analysis. Consider the software execution graph in Figure 4.3.6-1. Only the top level of the processing is illustrated here. The CPU time and

I/O requirements for each component are shown in Table 4.3.6-1.

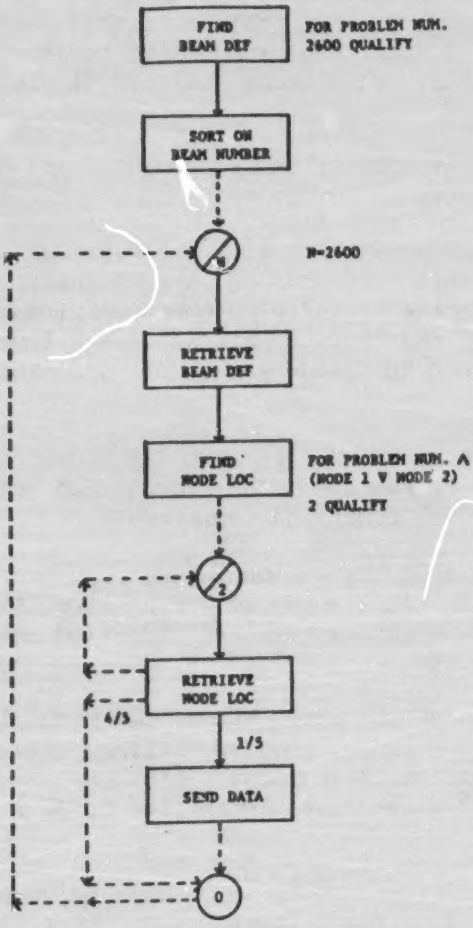

Figure 4.3.6-1 Optimization Example (reference (1))

Function	Disk Accesses	CPU Time(ms)
Find beam definition	7	111
Sort on beam number	72	32,644
Retrieve beam definition	72	88,332
Find node locations	21	3,018,726
Retrieve node locations	36	177,016
Send data	0	2,600
Total	208	3,319,929 ms.

TABLE 4.3.6-1 RESOURCE REQUIREMENTS FOR OPTIMIZATION EXAMPLE

The elapsed time to complete an I/O operation is assumed to be 30 ms. Other specifications are unimportant in this example.

The average response time for this scenario is 3326 seconds (55.4 minutes). This is clearly unacceptable for an interactive transaction. The bottleneck analysis indicates that the CPU is the critical resource since it has a higher ratio to the elapsed time than the I/O ratio. Furthermore, the "find node location" component is the critical component.

The processing details of this collapsed model are not shown; however, close examination of the details indicates that a "find" data base command is invoked for each of the three search keys, and then takes the intersection of the records that qualify. Also, it is found that the result of the "find" for the problem number search key is invariant throughout the loop and need not be repeated. A knowledge of the nature of the problem leads to the observation that most of the time (85%) the "find" on the node 1 key yields the same result as the "find" on the node 2 key from the previous pass through the loop, and need not be repeated. The results of this analysis indicate changes which optimize the process.

These optimizations are reflected in the execution graph in Figure 4.3.6-2. This graph is more complex, however, the total processing requirements are reduced, as shown in Table 4.3.6-2.

The response time has been reduced by 3023 seconds, a substantial savings! The response time (303 seconds) is still unacceptable for most on-line applications. Another optimization, storing the "beam def" data in beam number sequence, precludes the sort. The resulting response time is 269 seconds. This optimization process continues until a resulting response time of 82 seconds is obtained.

Function	Disk Accesses	CPU Time(ms)
Find beam definition	7	111
Sort beam number	72	32,644
Find node location	4	1,075
Retrieve beam definition	72	88,832
Find node location:		
B-tree I/O	17	102
Find 2 nodes	--	44,000
Retrieve 2 nodes	--	27,200
Find 1 node	--	26,000
Retrieve 1 node	--	71,800
Record I/O	36	216
Send data	0	2,600
Total	208	297,580 ms.

TABLE 4.3.6-2 RESOURCE REQUIREMENTS FOR REVISED OPTIMIZATION EXAMPLE

The performance is still only marginally acceptable, but it is a dramatic improvement over the original design. The bottlenecks are detected and corrected prior to actual coding and, therefore, the modifications require minimal effort.

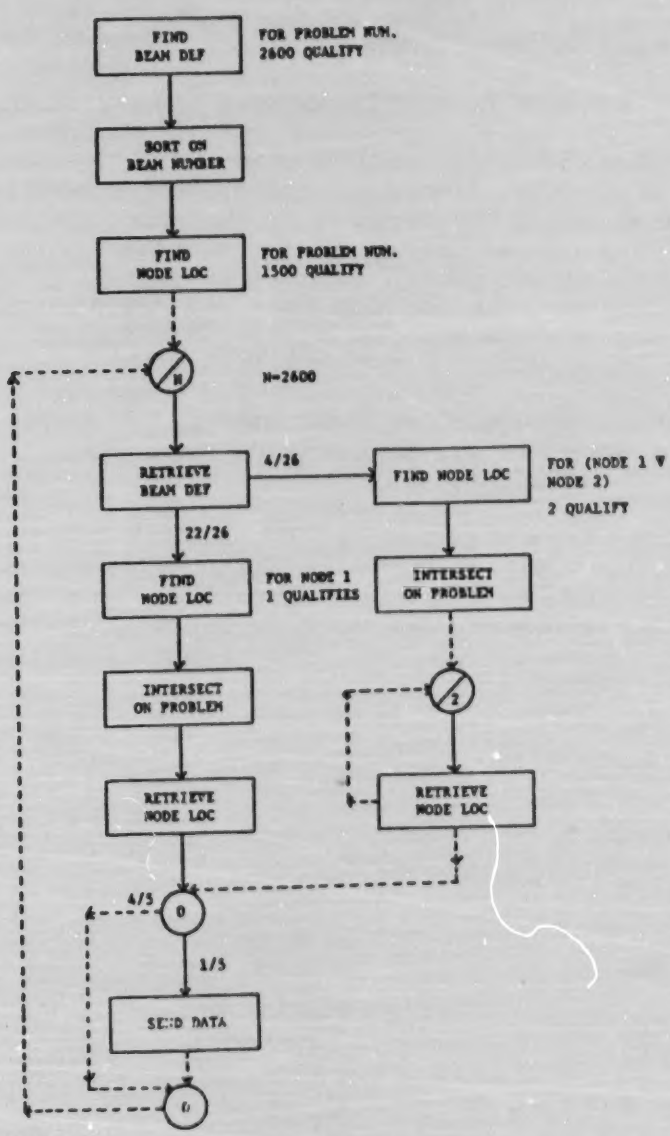

Figure 4.3.6-2 Revised Optimization Example (reference (1))

4.3.7. Effectiveness. The accuracy of the performance prediction is only as good as the quality of the performance specifications. The quality of the specifications usually improves during the design process. A simplified

approach is used to analyze queueing network models. This results in approximation of the relationships between contending resources. Several compensating features are used to offset the approximations used.

4.3.8. Applicability. The technique is generally applicable to nondistributed systems.

4.3.9. Learning. The user of this approach needs to be familiar with the intricacies of the modeling techniques used.

4.3.10. Costs. The preparation, analysis, and solution of the model costs approximately 5% to 15% of the total design costs.

4.3.11. References.

(1) SMITH, C.U., "The Prediction and Evaluation of the Performance of Software From Extended Design Specification", Ph.D. Dissertation, University of Texas at Austin, August 1980.

(2) SMITH, C.U., and BROWNE, J.C., "Performance Specifications and Analysis of Software Designs", Proceedings of the Conference on Simulation, Measurement, and Modeling of Computer Systems, Boulder, CO., August 1979.

4.4.1. Name. Assertion Generation

4.4.2. Basic features. Assertion generation is not so much a verification technique itself as it is foundational to a variety of other techniques. Assertion generation is the process of capturing the intended functional properties of a program in a special notation (called the assertion language) for insertion into the various levels of program specification, including the program source code. Other verification techniques utilize the embedded assertions in the process of comparing the actual functional properties of the program with the intended properties.

4.4.3. Information input. A specification of the desired functional properties of the program is the input required for assertion generation. For individual modules, this breaks down, at a minimum, to a specification of the conditions which are "assumed" true on a module entry and a specification of the conditions desired on module exit. If the specifications from which the assertions are to be derived include algorithmic detail, the specifications will indicate conditions which are to hold at intermediate points within the module as well. Additionally, assertions can state data characteristics, e.g. loop invariants, physical units or a variable, as input only(can not be set).

4.4.4. Information output. The assertions which are created from the functional or algorithmic specifications are expressed in a notation called the assertion language. This notation commonly includes higher level expressive constructs that are found, for example, in the programming language. An example of such a construct is a set. Most commonly, the assertion language is equivalent in expressive power to the first order predicate calculus. Thus, expressions such as "forall i in set S, A[i] A[i+1]" or "there exists x such that f(x) = 0" are possible. The assertions which are generated, expressing the functional properties of the program, can then be used as input to a dynamic assertion processor, a formal verification tool, walkthroughs, specification simulators, and inspections, among other V&V techniques.

4.4.5. Outline of method. Assertion generation proceeds hand-in-hand with the hierarchical elaboration of program functions. When, during development, a function is identified as being needed, it is usually first specified by what input it is expected to take and what the characteristics of the output are (outputs are often in terms of the input quantities). For such a function it is possible to generate input and output assertions without any knowledge of how the function performs its task. The input assertion expresses the requirements on the data the function is to use during its processing. The output assertion expresses what is to be true on function termination.

Later, as the function is elaborated, the designer or coder will identify the necessary steps to be taken in order to accomplish what is required of the function. After each step it can be said that a "part" of the task has been accomplished. That part is necessary for the proper operation of the next step, and so on, until the entire function has been realized. The character of each part can be captured by an assertion in the same way as the description of the entire function. The output assertion for one step represents (at least part of) the input assertion for the following step.

Such assertions are called intermediate assertions.

Each assertion, input, output, and intermediate is expressed using the assertion language and is placed into the specification of the function being implemented at the appropriate points. Thus, the program source text will include in it all the assertions developed during the requirements, design, and coding phases.

Some programming languages include facilities for expressing assertions in the source code but most do not. In such cases it is customary to include the assertions within comments, for indeed they are documentation expressing the desired functional characteristics of the program. Subsequent V&V tools, such as dynamic assertion processors, are constructed to utilize these special comments during their processing. Dynamic assertion processors are able to check the validity of the source assertions during program execution. Thus a method for dynamically verifying that the program is behaving according to its intended specification is possible.

For programs which contain loops (which is just about all programs), it is often important to formulate assertions which are always true at specific points within the loops. Such assertions are termed <u>invariant</u> or <u>inductive</u> assertions.

4.4.6. Example. Since assertion generation is so closely entwined with program development only a brief example is presented here. For more thorough examples see references (1-5).

During program development the requirement arises for sorting the elements of an array or table. In order to support flexible processing in the rest of the system, the array is declared with a large, fixed length. However, only a portion of the array has elements in it. The number of elements currently in the array, when passed to the sort routine, is contained in the first element of the array. The array is always to be sorted in ascending order. The sorted array is returned to the calling program through the same formal parameter.

The first specification of the sort routine may appear as:

```
      SUBROUTINE SORT ( A, DIM )
C
C     A is the array to be sorted
C     DIM is the dimension of A
C
C     sort array
C
      RETURN
      END
```

Figure 4.4.6-1 Sort Specification

The characteristics of the subroutine may be partially captured by the following assertions. Notationally, v="or" and &="and".

```
ASSERT INPUT (0≤A(1)≤DIM),(DIM≥2)
ASSERT OUTPUT (A(1)=0 v A(1)=1 & true) v
    (A(1)>1 & FORALL I IN [2 . . A(1)]  A(I)  A(I+1))
```

The input assertion notes the required characteristics of A(1) and DIM. The output assertion indicates that if there were 0 or 1 elements in the array, the array is sorted by default. If there are at least 2 elements in the array, then the array is in ascending order.

The next level of the program may have the following appearance. An intermediate assertion is now shown.

```
      SUBROUTINE SORT (A,DIM)
C
C     A is the array to be sorted
C     DIM is the dimension of A
C
      ASSERT INPUT (0≤A(1) DIM), (DIM>2)
      IF (A(1) .LE. 1) GOTO 100
      ASSERT (2≤A(1)≤DIM)
C
C     Sort non-trivial array
C
100   ASSERT OUTPUT (A(1)=0 v A(1)=1 & true) v
       (A(1)>1 & FORALL I IN [2 . . A(1)]  A(I)≤A(I+1))
      RETURN
      END
```

Figure 4.4.6-2 Sort Routine with Assertions

Suppose a straight selection sort algorithm is chosen for the non-trivial case (i.e., find the smallest element and place it in A(2), find the next smallest and place it in A(3), and so forth, where the original contents of A(I) is exchanged with the element that belongs in the Ith position in the sorted array). An appropriate intermediate assertion is included within the sorting loop.

```
C     PERFORM STRAIGHT SELECTION SORT
      DO 50 J = 2, A(1)
C
C       find smallest element in A(J) . .  A(A(1)+2)
C       let that element be A(K)
C       exchange A(J) and A(K)
C
      ASSERT (2≤J≤A(1))
      (FORALL I IN  [2 . . A(1)]   A(I)≤A(I+1))
 50   CONTINUE
```

Figure 4.4.6-3 Sort Routine with an Intermediate Assertion

A significant issue which we have not dealt with yet is asserting, on termination, that the sorted array is a permutation of the original array. In other words, we wish to assert that in the process of sorting, no elements were lost. To do this at the highest level, our first attempt at the program requires advanced assertion language facilities. The interested reader is referred to references (1) and (5).

4.4.7. Effectiveness. Assertion generation, particularly when used in conjunction with allied techniques like dynamic assertion processing or functional testing, can be extremely effective in aiding V&V. Such effectiveness is only possible, however, when the assertions are used to capture the important functional properties of the program. Assertions such as the following are of no use at all:

```
I = 0
I = I + 1
ASSERT I>0
```

Capturing the important properties can be a difficult process and is prone to error. Such effort is well rewarded, though, by increased understanding of the problem to be solved. Indeed, assertion generation is effective because the assertions are to be parallel to the program specifications. This parallelism enables the detection of errors, but effort is required.

A cost-effective procedure, therefore, is to develop intermediate assertions only for particularly important parts of the computation. Input assertions should always be employed, and output assertions whenever possible.

4.4.8. Applicability. The technique is generally applicable, in all development phases and for all programming languages.

4.4.9. Learning. Training and experience in writing assertions is the key to their effective use. Thoughtful consideration of the material contained in the references should enable a programmer to begin with useful assertions. Experience will sharpen the ability, especially if a dynamic assertion processor or other allied technique is also used.

4.4.10. Costs. Assertion generation is generally a manual technique, i.e., no machine resources are required. Effective use requires thoughtful problem and solution consideration, but no more than is normally required in professional task performance. Tools do exist that use symbolic execution to automatically generate loop invariant assertions. The cost then becomes that of symbolic execution.

4.4.11. References.

(1) TAYLOR, R.N., "Assertions in Programming Languages" SIGPLAN Notices, Vol. 15, 1, January 1980, pp. 105-114.

(2) MANNA, Z; WALDING, R., "The Logic of Computer Programming" IEEE-TSE, SE-4, 3, May 1978, pp. 199-229 (especially pages 199-204).

(3) HOARE, C.A.R., "Proof of a Program: FIND" CACM,V. 14, 1, January 1971, pp. 39-45.

(4) HETZEL, W. D. ed., "Program TEST Methods",1973, Articles on pages 7-10, 17-28, 57-72.

(5) CHOW, T. S., "A Generalized Assertion Language", Proceedings of the Second International Conference on Software Engineering,San Francisco, California, pp. 392-399.

(6) STUCKI, L. G., and FOSHEE, G. L., "New Assertion Concepts for Self-Metric Software", Proceedings of the 1975 Conference on Reliable Software,pp. 59-71.

4.5.1. Name. Assertion Processing.

4.5.2. Basic Features. Assertion processing is the process whereby the program's assertions (containing user specific assertions as described in the previous section) are checked during program execution. As such, the techniques serve as a bridge between the more formal program correctness proof approaches and the more common "black box" testing approaches.

4.5.3. Information Input. Information input to this technique consists of a program which contains the assertions to be processed. The program can be written in any language but may be restricted to a particular language if an automatic tool is used to perform the dynamic assertion processing. Moreover, if a tool is used, the format for specifying the assertions will be that defined by the particular tool. Generally, assertions are specified as comments in the source program.

4.5.4. Information Output. Output from a dynamic assertion process normally consists of a list of the assertion checks which were performed and a list of exception conditions with trace information for determining the nature and location of the violations.

4.5.5. Outline of Method. The assertions are generated by the developer as described in the "Assertion Generation" technique in the previous section. The assertions are then translated into host language program statements which actually perform the assertion checking at program execution time. The translation can be done manually or through the use of an automated dynamic assertion processor.

The translation process is shown in the following illustration. An assertion of the form:

```
(* ASSERT condition*)

is translated into:

IF NOT (condition) THEN

     Process assertion violation;
```

The processing of the assertion violation will, minimally, keep track of the total number of violations for each assertion, print a message indicating that a violation of the assertion has occurred, and print the values of the variables referenced in the assertion. In addition, the location, i.e. statement number, and the number of times the assertion is checked may be kept and printed when a violation occurs.

Sufficient information should be reported upon violation of an assertion to assist the programmer of the specific nature of the error.

An automated dynamic assertion processor can be of great assistance by alleviating for the programmer the burden of hand generating the source code necessary to perform the assertion checking. Not only will this save time but

it will also perform the translation more reliably.

Specifying assertions within comments is a valuable form of documentation and also ensures that the source program is kept free of non-portable, tool specific directives.

It is important to note that dynamic assertion processing for non-real time programs must not alter the functional behavior of a program. Use of a good automated tool will ensure this. Execution time, however, will be increased; the amount of which will depend on the number of assertions which are processed. It is important to note that dynamic assertion processing can alter the functional behavior of a program by altering the execution timing.

In order to effectively utilize assertion processing, test data should be generated which will cause the execution of each assertion.

4.5.6. Example. The program segment in Figure 4.5.6-1 is taken from a Pascal program which calls on routine 'sort' to sort array 'A', consisting of 'N' integer elements, in ascending order. The assertion following the call to sort asserts that the elements are indeed in ascending order upon return from the sort procedure. The numbers to the left are the line numbers from the original source.

```
                .
                .
12   Var
13              N : integer;
14              A : array [1..MAXN] of integer;
                .
                .

26   begin
                .
                .

56              sort (N,A);
57   (* assert forall i in [1..N-1]:A[i]<= A[i+1] *);
```

Figure 4.5.6-1 Source Program with Untranslated Assertion

The program segment in Figure 4.5.6-2 is that which results after all of the assertions have been translated into Pascal. Note that a rather large number of statements were used to implement the assertion. This is due to the rather involved checking required to implement an "assert forall. . .". Simpler assertions will require fewer statements. The spec could be reduced through the use of a common assertion violation procedure.

```
                .
                .
12   Var
13              N : integer;
14              A : array [1..MAXN] of integer;
15              AssertVioCount : array [1.. NumofAsserts] of integer;
16              AssertXqtCount : array [1.. NumofAsserts] of integer;
17              assert : boolean;
                .
                .

29   begin
                .
                .
77              sort (N,A);
78              (* assert forall: in [1..N-1] : A[i] =A[i+1] *);
79                 AssertXqtCount[3]:= AssertXqtCount[3]+1;
80                 assert : = true;
81                 i := 1 ;
82               while(i<= N) and(assert) do(* check assertion *)
83                  if A[i]>A[I+1] then
84                        assert := false
85                  else
86                        i := i + 1;
87               if not assert then begin(* assertion violation *)
88                    AssertVioCount[3] := AssertVioCount[3] = 1;
89                    Writeln ('violation of assertion 3 at statement 57 ');
90                    Writeln ('on execution:', AssertXqtCount[3]);
91                    Writeln ('arrayA = ', A)
92               end (* assertion violation *);
                   .
                   .
```

Figure 4.5.6-2 Source Program with Translated Assertions

During the testing the following values of A were used in successive executions of the sort routine.

execution	array A									
1	0	3	12	27	53	171	201	251	390	501
2	0	12	3	53	27	201	171	390	251	501
3	501	390	251	201	171	53	27	12	3	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	100	100	100	999	999	999	1000

The resulting execution produced the following assertion violation:

violation of assertion 3 at statement 57 on execution: 3
array A = 3 12 27 53 171 201 251 340 501 0

This was the only violation which occurred.

Subsequent analysis of the sort procedure indicated that the error was due to an "add-by-one" error on a loop limit.

4.5.7. Effectiveness. The effectiveness of dynamic assertion processing will depend upon the quality of the assertions included in the program being analyzed. Moreover, if the translation is being done by hand, the amount of time required to translate, coupled with the unreliability associated with the process will reduce its effectiveness. Nevertheless, the technique can be of significant value in revealing the presence of program errors.

4.5.8. Applicability. The technique is generally applicable.

4.5.9. Learning. A functional understanding of assertions is all that is necessary in order to manually use this technique. If a tool is used, then an hour or so should be sufficient to learn the specification syntax for asssertions acceptable to that tool. Of course, the generation of useful assertions (see "Assertion Generation" writeup) is necessary in order for this technique to be truly valuable.

4.5.10. Costs. The costs associated with this technique are almost entirely comprised of the amount of time required to translate the assertions into source code. If done manually, this could amount to significant cost. If done automatically, the cost will be on the order of compilation (Assertion Processors are usually implemented as source language preprocessors). If a tool is not available, it may well be worth the cost to develop one in-house.

4.5.11. References.

(1) STUCKI, L. G., and FOSHEE, G. L., "New Assertion Concepts for Self-Metric Software", Proceedings, 1975 Conference on Reliable Software, pp.59-71.

(2) ANDREWS, D.M., "Using Executable Assertions for Testing", 13th Annual Asilomar Conference on Circuits, Systems, and Devices, Nov. 1979.

4.6.1. Name. Cause-Effect Graphing.

4.6.2. Basic features. Cause-effect graphing is a test case design methodology. It is used to select in a systematic manner a set of test cases which have a high probability of detecting errors that exist in a program. This technique explores the inputs and combinations of input conditions of a program in developing test cases. It is totally unconcerned with the internal behavior or structure of a program. In addition, for each test case derived, the technique identifies the expected outputs. The inputs and outputs of the program are determined through analysis of the requirement specifications. These specifications are then translated into a Boolean logic network or graph. The network is used to derive test cases for the software under analysis.

4.6.3. Information input. The information that is required as input to carry out this technique is a natural language specification of the program that is to be tested. The specification should include all expected inputs and combinations of expected inputs to the program, as well as expected outputs.

4.6.4. Information output. The information output by the process of cause-effect graphing consists of the following:

a. An identification of incomplete or inconsistent statements in the requirement specifications.

b. A set of input conditions on the software (causes).

c. A set of output conditions on the software (effects).

d. A Boolean graph that links the input conditions to the output conditions.

e. A limited entry decision table that determines which input conditions will result in each identified output condition.

f. A set of test cases.

g. The expected program results for each derived test case.

The above outputs represent the result of performing the various steps recommended in cause-effect graphing.

4.6.5. Outline of method. A cause-effect graph is a formal language translated from a natural language specification. The graph itself is represented as a combinatorial logic network. The process of creating a cause-effect graph to derive test cases is described briefly below.

a. Identify all requirements of the system and divide them into separate identifiable entities.

b. Carefully analyze the requirements to identify all the causes and effects in the specification. A cause is a distinct input condition; an effect is an output condition or system transformation (an effect that an input has on the state of the program or system).

c. Assign each cause and effect a unique number.

d. Analyze the semantic content of the specification and transform it into a Boolean graph linking the causes and effects; this is the cause-effect graph.

- o Represent each cause and effect by a node identified by its unique number.
- o List all the cause nodes vertically on the left side of a sheet of paper; list the effect nodes on the right side.
- o Interconnect the cause and effect nodes by analyzing the semantic content of the specification. Each cause and effect can be in one of two states: true or false. Using Boolean logic, set the possible states of the causes and determine under what conditions each effect will be present.
- o Annotate the graph with constraints describing combinations of causes and/or effects that are impossible because of syntactical or environmental constraints.

e. By methodically tracing state conditions in the graph, convert the graph into a limited entry decision table as follows. For each effect, trace back through the graph to find all combinations of causes that will set the effect to be true. Each such combination is represented as a column in the decision table. The state of all other effects should also be determined for each such combination. Each column in the table represents a test case.

f. Convert the columns in the decision table into test cases.

This technique to create test cases has not yet been totally automated. However, conversion of the graph to the decision table, the most difficult aspect of the technique, is an algorithmic process which could be automated by a computer program.

4.6.6. Example. A database management system requires that each file in the database have its name listed in a master index which identifies the location of each file. The index is divided into ten sections. A small system is being developed which will allow the user to interactively enter a command to display any section of the index at his terminal. Cause-effect graphing is used to develop a set of test cases for the system.

a. The specification for this system is as follows:
To display one of the ten possible index sections, a command must be entered consisting of a letter and a digit. The first character entered must be a D (for display) or an L (for list) and it must be in column 1. The second

character entered must be a digit (0-9) in column 2. If this command occurs, the index section identified by the digit is displayed on the terminal. If the first character is incorrect, error message A is printed. If the second character is incorrect, error message B is printed. The error messages are:

A: INVALID COMMAND
B: INVALID INDEX NUMBER

b. The causes and effects have been identified as follows. Each has been assigned a unique number.

Causes
1. Character in column 1 is D.
2. Character in column 1 is L.
3. Character in column 2 is a digit.

Effects
50. Index section is displayed.
51. Error message A is displayed.
52. Error message B is displayed.

c. Figure 4.6.6-1, a Boolean graph, is constructed through analysis of the semantic content of the specification.

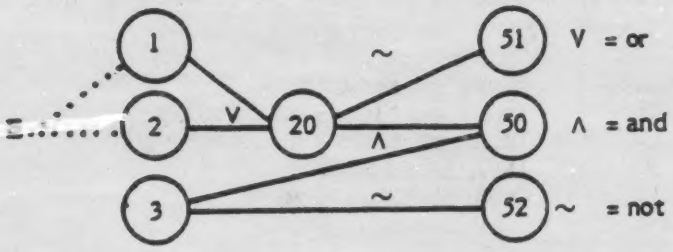

Figure 4.6.6-1 Boolean Graph

Node 20 is an intermediate node representing the Boolean state of node 1 or node 2. The state of node 50 is true if the state of nodes 20 and 3 are both true. The state of node 20 is true if the state of node 1 or node 2 is true. The state of node 51 is true if the state of node 20 is not true. The state of node 52 is true if the state of node 3 is not true.

Nodes 1 and 2 are also annotated with a constraint that states that causes 1 and 2 cannot be true simultaneously (the Exclusive constraint).

d. The graph is converted into a decision table, figure 4.6.6-2. For each test case, the bottom of the table indicates which effect will be present (indicated by a 1). For each effect, all combinations of causes that will result in the presence of the effect is represented by the entries in the columns of the table. Blanks in the table mean that the state of the cause is

irrelevant.

	Test Cases			
Causes	1	2	3	4
1	1	0	0	
2	0	1	0	
3	1	1		0
Effects				
50	1	1	0	0
51	0	0	1	0
52	0	0	0	1

Figure 4.6.6-2 Decision Table

e. Each column in the decision table is converting into test cases, figure 4.6.6-3.

Test Case #	Inputs	Expected Results
1	D5	Index section 5 is displayed
2	L4	Index section 4 is displayed
3	B2	INVALID COMMAND
4	DA	INVALID INDEX NUMBER

Figure 4.6.6-3 Test Cases

4.6.7. Effectiveness. Cause-effect graphing is a technique used to produce a useful set of test cases. It also has the added capability of pointing out incompleteness and ambiguities in the requirement specification. However, this technique does not produce all the useful test cases that can be identified. It also does not adequately explore boundary conditions.

4.6.8. Applicability. Cause-effect graphing can be applied to generate test cases in any type of computing application where the specification is clearly stated and combinations of input conditions can be identified. Manual application of this technique is a somewhat tedious, long, and moderately complex process. However, the technique could be applied to selected modules where complex conditional logic must be tested.

4.6.9. Learning. Cause-effect graphing is a mathematically-based technique that requires some knowledge of Boolean logic. The requirement specification of the system must also be clearly understood in order to successfully carry out the process.

4.6.10. Costs. Manual application of this technique will be highly labor intensive.

4.6.11. References.

(1) ELMENDORF, W.R., "Cause-Effect Graphs in Functional Testing," IBM Systems Development Division, TR-00.2487, Poughkeepsie, New York, 1973.

(2) MYERS, GLENFORD, "The Art of Software Testing," Wiley-Interscience, New York, 1975.

(3) MYERS, GLENFORD, "Software Reliability: Principles and Practices," Wiley-Interscience, New York, 1976.

4.7.1. Name. Code Auditor.

4.7.2. Basic features. A code auditor is a computer program which is used to examine source code and automatically determines whether prescribed programming standards and practices have been followed.

4.7.3. Information input. The information input to a code auditor is the source code to be analyzed and the commands necessary for the code auditor's operation.

4.7.4. Information output. The information that is output by a code auditor is a determination of whether the code being analyzed adheres to prescribed programming standards. If errors exist, information is generated detailing which standards have been violated and where the violations occur. This information can appear as error messages included with a source listing or as a separate report. Other diagnostic information, such as a cross-reference listing, may also be output as an aid in making the needed corrections.

4.7.5. Outline of method. Code auditors are fully automated tools which provide an objective, reliable means of verifying that a program complies with a specified set of coding standards. Some common programming conventions that code auditors can check for are given below.

- o Correct syntax - Do all program statements conform to the specifications of the language definition?
- o Portability - Is the code written so that it can easily operate on different computer configurations?
- o Use of structured programming constructs - Does the code make proper use of a specified set of coding constructs such as IF-THEN-ELSE or DO-WHILE?
- o Size - Is the length of any program unit not more than a specified number of statements?
- o Commentary - Is each program unit appropriately documented; e.g., is each unit preceded by a block of comments which indicates the function of the unit and the function of each variable used?
- o Naming conventions - Do the names of all variables, routines, and other symbolic entities follow prescribed naming conventions?
- o Statement labeling - Does the numeric labeling of statements follow an ascending sequence throughout each program unit?
- o Statement ordering - Do all statements appear in a prescribed order; e.g., in a Fortran program, do all FORMAT statements appear at the end and DATA statements before the first executable statement of a routine?
- o Statement format - Do all statements follow a prescribed set of formatting rules which improve program clarity; e.g., are all DO-WHILE loops appropriately indented?

As demonstrated by this list, code auditors vary in sophistication according to their function. Each auditor, however, requires some form of syntax analysis to be performed. Code must be parsed by the auditor and given an internal representation suitable for analysis. Because this type of processing is found in many static analysis tools, many code auditors are part of a more general tool having many capabilities. For example, a compiler is a form of code auditor that checks for adherence to the specifications of a language definition. PFORT, a tool used to check Fortran programs for adherence to a portable subset of American National Standard Institute (ANSI) Fortran 66, also has the capability of generating a cross-reference listing.

Code auditors are useful to programmers as a means of self-checking their routines prior to turnover for integration testing. These tools are also of value to software product assurance personnel during integration testing, prior to formal validation testing, and again prior to customer delivery.

4.7.6. Example.

a. Application. A flight control program is to be coded entirely in PFORT, a portable subset of ANSI Fortran 66. The program is to be delivered to a military government agency, which will install the software on various computer installations. In addition, the customer requires that each routine in the program be clearly documented in a prescribed format. All internal program comments are to be later compiled as a separate source of documentation for the program.

b. Error. A named common block occurs in several routines in the program. In one routine, the definition of a variable in that block has been omitted because the variable is not referenced in that routine. This is, however, a violation of a rule defined in PFORT, which requires that the total length of a named common block agree in all occurrences of that block.

c. Error discovery. A code auditor which checks Fortran for adherence to PFORT detects this error immediately. The programmer of this routine is informed that the routine is to be appropriately modified and that any confusion over the use of the variable is to be clarified in the block of comments that describe the function of each defined variable in the routine. A code auditor that checks for the presence of appropriate comments in each routine is used to verify that the use of the variable is appropriately documented. At the end of code construction, all such internal program documentation will be collated and summarized by another code auditor which processes machine readable documentation imbedded in source code.

4.7.7. Effectiveness. Code auditors are very effective tools in certifying that software routines have been coded in accordance with prescribed standards. They are much more reliable than manually performed code audits and are highly cost effective as they are less time consuming than manual audits.

4.7.8. Applicability. Code auditors can be generally applied to any type of source code. However, each specific tool will be language dependent (i.e., will operate correctly only for specified source languages), and will only

accept input that appears in a prescribed format.

4.7.9. Learning. No special training is required to use code auditors. As code auditors may be used by a wide variety of people (programmers, managers, quality assurance personnel, customers), ease in their use is an important attribute. In order to use code auditors effectively, however, some learning is required to gain familiarity with the standards upon which the auditor is based.

4.7.10. Costs. Code auditors are generally very inexpensive to use as their overhead is usually no more than the cost of a compilation.

4.7.11. References.

(1) BROWN, J.R. and FISCHER, K., "A Graph Theoretic Approach to the Verification of Program Structures," Proceedings of the 3rd International Conference on Software Engineering, May 1978.

(2) RYDER, B.G., and HALL, A.D., "The PFORT Verifier," Computing Science Technical Report #12, Bell Laboratories, Murry Hill, New Jersey, March 1975.

(3) FISCHER, K.F., "User's Manual for Code Auditor, Code Optimizer Advisor, Unit Consistency Analyses," TRW Systems Group, Redondo Beach, California, July 1974.

(4) HOPKINS, T.R., "PBASIC- A Verifier for BASIC," Software Practice and Experience, Vol 10, pp. 175-181, 1980.

4.8.1. Name. Comparators.

4.8.2. Basic features. A comparator is a computer program used to compare two versions of source data to establish that the two versions are identical or to specifically identify where any differences in the versions occur.

4.8.3. Information input. Input to comparators consists of two versions of source data to be compared and those commands necessary for the comparator to operate. The source data may be:

a. Source programs
b. Sets of program test cases or test results
c. Databases
d. Arbitrary data files

Many comparators provide various user options, such as whether blank lines are to be included in compare processing, to control comparison operation.

4.8.4. Information output. The output from a comparator is a listing of the differences, if any, between the two versions of input. Various report writing options are usually supplied by the comparator to designate the desired format of the output, e.g., whether each difference found should be preceded by line numbers. Many general comparator utility programs installed in large text-editing systems can also create a file of text-editor directives that can be used to convert one input file into the other.

4.8.5. Outline of method. Comparators are fully automated tools which serve to eliminate the tedious, time-consuming task of performing large numbers of comparisons. They are most useful during program development and maintenance. During program development, they provide a means of ensuring that only the intended portions of a program are changed when modifications are to be made to the latest version. When regression testing must be performed following software corrections or updates, comparators provide an efficient means of comparing current test cases and test results with past ones.

Comparators are widely available and are often provided as general utilities in operating systems. Other comparators may be more specialized and require input files to be of a prescribed format in order for the tool to operate correctly.

Comparators are invaluable tools in assisting configuration management and change control as the software takes different forms during development.

4.8.6. Example.

a. Application. A large command and control flight software system is being developed. During system testing, the generation of many different databases is required as a source of input data for each associated test case. Strict control of the databases, including identification of their similarities and differences, must constantly be maintained in order to properly verify test results.

b. Error. A bug in the software causes the execution of Test Case 3 to generate test results which are totally incompatible with the results of Test Case 1, though the input in both test cases is almost identical.

c. Error discovery. A comparator was used to compare the databases used in Test Case 1 and 3. The location of specific differences in the two files determined exactly which input data should be examined more closely and when traced through the program the error was found.

4.8.7. Effectiveness. Comparators are most effective during software testing and maintenance when periodic modifications to the software are anticipated. Their overall effectiveness is dependent upon the quality of their use.

4.8.8. Applicability. This method is generally applicable.

4.8.9. Learning. A minimal amount of effort is required to learn how to use comparators effectively. The tool's user documentation should provide sufficient information for its proper utilization.

4.8.10. Costs. Comparators are generally inexpensive to use. Their cost is similar to that of performing two passes of read operations on one file.

4.8.11. References.

(1). HETZEL, William, "Program Test Methods", Prentice-Hall, Inc., 1973.

(2). DEC IAS/RSX-11 "Utilities Procedure Manual", Digital Equipment Corporation, 1978.

4.9.1. Name. Control Structure Analyzer.

4.9.2. Basic features. Application of an automated structure analyzer to either code or design allows detection of some types of improper subprogram usage and violation of control flow standards. It also identifies control branches and paths used by test coverage analyzers. A structure analyzer is also useful in providing required input to data flow analyzers and is related in principle to code auditors.

4.9.3. Information input. Two input items are required by a structure analyzer. The first is the text of the program or design to be analyzed. Typically the text is to be provided to the analyzer in an intermediate form, i.e., after scanning and parsing, but not as object code. Often structure analyzers are incorporated within compilers.

The second input item is a specification of the control flow standards to be checked. These standards are often completely implicit in that they may be part of the rules for programming in the given language or design notation. An example of such a rule is that subprograms may not be called recursively in FORTRAN. Individual projects may, however, establish additional rules for internal use. Many such rules, for instance limiting the number of lines allowed in a subprogram, can be checked by a code auditor. Others, however, can require a slightly more sophisticated analysis and are therefore performed by a structural analyzer. Two examples in this category are "All control structures must be well nested" and "Backward jumps out of control structures are not allowed."

Typically this second input item is not directly supplied to a structure analyzer, but is incorporated directly in the tool's construction. Therefore, substantial inflexibility is common.

4.9.4. Information output. Error reports and a program call graph are the most common output items of a structure analyzer. Error reports indicate violations of the standards that were input to the tool. Call graphs indicate the structure of the graph with respect to the use of subprograms; associated with each subprogram is information indicating all routines which call the subprogram and all routines which are called by it. The presence of cycles in the graph (A calls B calls A) indicate possible recursion. Routines which are never called are evident, as well as attempts to call nonexistent routines.

In checking adherence to control flow standards, the structure analyzer may also output a flow graph for each program unit. The flow graph represents the structure of the program with each control path in the program represented by an edge in the graph. Additionally, structurally "dead" code within each module is detectable.

The flow graph and the call graph are items required as input by data flow analyzers, and it is common for the two analysis capabilities to be combined in a single automated tool.

4.9.5. Outline of method. Since structure analysis is an automated static analysis technique, little user action is required. Aside from providing the input information, the user is only required to peruse the output reports and determine if program changes are required. Some simple manifestations of the tool may not provide detailed analysis reports; therefore, more responsibility is placed on the user to examine, for example, the call graph for the presence of cycles.

4.9.6. Example.

a. An online management information system program, figure 4.9.6-1, calls a routine MAX to report the largest stock transaction of the day for a given issue. If MAX does not have the necessary information already available, RINPUT is called to read the required data. Since RINPUT reads many transactions for many issues, a sort routine is utilized to aid in organizing the information before returning it to the calling routine. Due to a keypunch error the sort routine calls routine MAX (instead of the proper routine MAXI) to aid in the sorting process. This error will show up as a cycle in the call graph and will be reported through use of a structure analyzer.

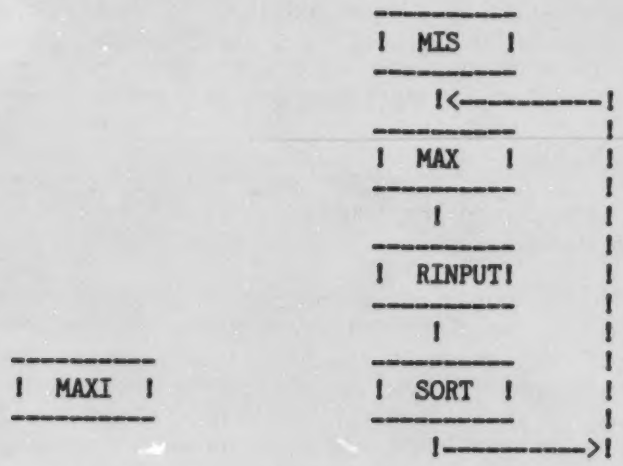

Figure 4.9.6-1 MIS Flow Chart

b. As part of the programming standards formulated for a project, the following rule is adopted:

"All jumps from within a control structure must be to locations after the end of the structure."

Figure 4.9.6-2, a segment of Pascal code, contains a violation of this rule which would be reported by a suitably constructed structure analyzer.

```
100 :    X: = 100;
while X>70 do
     begin
       .
       .
       .
     if Z = 5 then goto 100;
       .
       .
       .
     end;
```

Figure 4.9.6-2 Goto Violation

4.9.7. Effectiveness. The technique is completely reliable for detecting violations of the standards specified as input. The standards, however, only cover a small range of programming standards and possible error situations. Thus, the technique is useful only in verifying very coarse program properties. The technique's prime utility, therefore, is in the early stages of debugging a design or code specification.

4.9.8. Applicability. The technique is generally applicable and may be applied in design and coding phases. Particular applicability is indicated in systems involving large numbers of subprograms and/or complex program control flow.

4.9.9. Learning. Minimal training is required for use of the technique. See "Outline of Method."

4.9.10. Cost. Little human cost is involved as there is no significant time spent in preparing the input or interpreting the output. For an average program, computer resources are small ;the processing required can be done very efficiently and only a single run is required for analysis. For large or complex programs, the cost can be quite high. A plotter, which produces the most readable structure diagrams, drives the cost up.

4.9.11. References.

(1) FAIRLEY, Richard E., "Tutorial: Static Analysis and Dynamic Testing of Computer Software," Computer,Vol. 11, No. 4, pp. 14-23, April, 1978.

(2) HOWDEN,W.E.,"Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering,vol. SE-2,no. 3,1976.

4.10.1. Name: Cross-Reference Generators

4.10.2. Basic features. Cross-reference generators produce lists of data names and labels showing all of the places they are used in a program.

4.10.3. Information input. Input to cross-reference generators consists of a computer program in either source or object format.

4.10.4. Information output. Output from a cross-reference generator is an alphabetized list of variable names, procedure names and statement labels showing the locations in the program where they are defined and referenced. Other information, which is sometimes included, is data type, attributes, and usage information.

4.10.5. Outline of Method. Cross-reference generators provide useful information which can aid both program development and maintenance. They aid program development by helping identify errors such as misspelled identifiers and improperly typed variables. Program maintenance is aided by helping to locate, by variable or statement label, those portions which may be affected by a program change (e.g., a variable name needs to be changed).

Cross-reference generators are widely available and are usually provided with program source text analyzers such as compilers, standards checkers and data flow analyzers.

Cross-reference listings should be checked in detail after a program change has been made to check for misspelled identifiers and incorrect usage, etc.

4.10.6. Example.

a. Application. A communication network controller manages the control of a network of high-speed communication lines connecting a large number of CRT terminals to an airline reservation system computer.

b. Error. A variable used to store message addresses is assigned an address which erroneously points to a location storing highly critical queue control information. A subsequent call to the device handler causes data to be read into the critical storage area causing a system crash.

c. Error discovery. A quick study of software's cross-reference listing showed all the locations where the offending variable was used, one of which clearly showed that the error was due to improper use of a pointer variable.

Figure 4.10.6-1 shows a sample program listing and corresponding cross-reference list. The program is a utility routine used by a large aerodynamic analysis program. The tool which generated the report is called PFORT (2) which performs various FORTRAN source analyses. The list shows for each identifier its type (e.g., integer or real), usage (e.g., variable or function), attributes (e.g., argument, whether the variable has been set, scalar or array) and the line numbers where it is referenced.

```
PFORT VERIFIER 3/15/75 VERSION

      C
      C DRIVER PROGRAM TO TEST EUCLIDEAN NORM FUNCTION
      C
1             INTEGER X(100)
2             LOGICAL ERR
3             COMMON/ERROR/ERR
4     1       READ(5,10) I
5     10      FORMAT(I2,I5)
      C       END OF DATA CHECK
6             IF(I.GT.100) STOP
7             READ(5,10) (X(J), J=1,I)
8             ERR=.FALSE.
9             ANS=ENORM(I, X)
10            IF (.NOT.ERR) GOTO 2
11            WRITE (6,20)
12    20      FORMAT (15H BAD VALUE OF N)
13            GOTO 1
14    2       WRITE (6,30) ANS
15    30      FORMAT (6H NORM=,E15.7)
16            GOTO 1
17            END

PROGRAM UNIT   *MAIN

NAME      TYPE     USE     ATTRIBUTES      REFERENCES

ANS       R        V             SS        9   14
ENORM     R        FN                      9
ERR       EL       V       C     SS        2    3    8   10
I         I        V             SS        4    6    7    9
J         I        V             SS        7
X         EI       V             SA1       1    7    9
10                                         4    5    7
1                                          4   13   16
20                                        11   12
2                                         10   14
30                                        14   15

COMMON BLOCKS
ERROR ERR
```

Figure 4.10.6-1 Sample Cross-Reference Examples

Key to Figure 4.10.6-1

Type Key
column 1:

E explicitly typed

column 2:

I INTEGER
R REAL
D DOUBLE PRECISION
C COMPLEX
L LOGICAL
H HOLLERITH

Use Key
columns 1, 2:

FA arithmetic-statement function argument
FN function name
E external (function or subroutine)

GT assigned goto variable
IF intrinsic function
SF arithmetic statement function
SN subroutine name
V variable

Attribute Key

column 1:
C in COMMON

column 2:
E in an EQUIVALENCE statement

column 3:
A dummy argument

column 4:
S value set by program unit

column 5, 6:
S scalar
An array with n dimensions

Figure 4.10.6-1 Sample Cross-Reference Examples (Continued)

4.10.7. Effectiveness. Cross-reference generators are most effective during the software maintenance phase to help determine where software errors are occurring, as seen in the previous example. Cross-reference generators are tools whose utility can often be taken for granted or even considered bothersome (e.g., "it produces too much paper"). Its lack of availability, however, will painfully demonstrate how necessary this seemingly basic capability is. Nevertheless, its true effectiveness is totally dependent upon the quality of its use.

4.10.8. Applicability. This method is generally applicable.

4.10.9. Learning. Minimal effort is required to learn how to effectively utilize cross reference generators.

4.10.10. Costs. Cross-reference programs are widely available, usually as a function provided by a larger system (e.g., a compiler) and add only an incremental amount to the total cost.

4.10.11. References.

(1) RYDER, B.G. and HALL A.D., "The PFORT Verifier," Computing Science Technical Report, No.12, Bell Labs, March, 1975.

4.11.1. Name. Data Flow Analyzers.

4.11.2. Basic features. Data flow analyzers are tools which can determine the presence or absence of data flow errors; that is, errors that are represented as particular sequences of events in a program's execution. The following description is limited to sequential analyzers although efforts are under way to include synchronous and concurrent events.

4.11.3. Information input. Data flow analysis algorithms operate on annotated graph structures which represent the program events and the order in which they can occur. Specifically, two types of graph structures are required: a set of annotated flowgraphs and a program invocation (or call) graph. There must be one flowgraph for each procedure. A flowgraph is a digraph whose nodes represent the execution units (usually statements) of the procedures, and whose edges are used to indicate the progression of execution units. Each node is annotated with indications of which program events occurred as a consequence of its execution. The program invocation (call) graph is also a digraph whose purpose is to indicate which procedures can invoke which others. Its nodes represent the procedures of the program and its edges represent the invocation relation.

4.11.4. Information output. The output of data flow analysis is a report on the presence of any specified event sequences in the program. If any such sequences are present, then the identity of each sequence is specified and a sample path along which the illegal sequence can occur is used. The absence of any diagnostic message concerning the presence of a particular event sequence is a reliable indicator of the absence of that sequence.

4.11.5. Outline of method. Data flow analyzers rely basically upon algorithms from program optimization to determine whether any two particular specified events can occur in sequence. Taking as input a flowgraph annotated with all events of interest, these algorithms focus upon two events and determine: 1) whether there exists some program path along which the two occur in sequence, and 2) whether on all program paths the two must occur in sequence. If one wishes to determine illegal event sequences of length three or more, these basic algorithms can be applied in succession.

A major difficulty arises in the analysis of programs having more than one procedure, because the procedure flowgraphs often cannot be completely annotated prior to data flow analysis. Flowgraph nodes representing procedure invocations must be left either partially or completely unannotated until the flowgraphs of the procedures which they represent have been analyzed. Hence, the order of analysis of the program's procedures is critical. This order is determined by a postorder traversal of the invocation graph in which the bottom level procedures are visited first, then those which invoke them, and so forth until the main level procedure is reached. For each procedure, the data flow analysis algorithms must determine the events which can possibly occur both first and last and then make this information available for annotation of all nodes representing invocations of this procedure. Only in this way can it be assured that any possible illegal event sequence will be determined.

4.11.6. Example. The standard example of the application of data flow analysis is to the discovery of references to uninitialized program variables. In this case, the program events of interest are the definition of a variable, the reference to a variable, and the omission of a definition of a variable. Hence, all procedure flowgraphs are annotated to indicate which specific variables are defined, referenced, and undefined at which nodes. Data flow analysis algorithms are then applied to determine whether the definition omission event can be followed by the reference event for any specific variable without any intervening definition event for that variable. If so, a message is produced indicating the possibility of a reference to an uninitialized variable and a sample program path along which this will occur. A different algorithm is also used to determine if a specific variable definition omission must, along all paths, be followed by reference without intervening definition. For invoked procedures, these algorithms are also used to identify which parameters and global variables are sometimes used and always used as inputs and outputs. This information is used to annotate all nodes representing the invocation of this procedure, to enable analysis of these higher level procedures.

Data flow analysis might also be applied to the detection of illegal sequences of file operations in programs written in languages such as COBOL. Here the operations of interest would be opening, closing, defining (i.e., writing), and referencing (i.e., reading) a file. Errors whose presence or absence could be determined would include: attempting to use an unopened file, attempting to use a closed file, and reading an empty file.

4.11.7. Effectiveness. As noted, this technique is capable of determining the absence of event sequence errors from a program, or their presence in a program. When an event sequence error is detected, it is always detected along some specific path. Because these techniques do not study the executability of paths, the error may be detected on an unexecutable path and hence give rise to a spurious message. Another difficulty is that this technique is unreliable in distinguishing individual elements of an array. Hence, arrays are usually treated as if they were simple variables. As a consequence, illegal sequences of operations on specific array elements may be overlooked.

4.11.8. Applicability. Data flow analyzers can be applied to any annotated graph. Therefore, the availability of this technique is only limited and restricted by the availability of the (considerable) tools and techniques needed to construct such flowgraphs and call graphs.

4.11.9. Learning. This technique requires only a familiarity with and understanding of the output messages. No input data or user interaction is required.

4.11.10. Costs. This technique requires computer time, but the algorithms employed are highly efficient, generally executing in time which is linearly proportional to program size. Experience has shown that the construction of the necessary graphs can be a considerable cost factor, however. Potential users are warned that prototype tools exploiting this technique have proven quite costly to operate.

As noted above, no human input or interaction is required, resulting in only the relatively low human cost for interpretation of results.

4.11.11. References.

(1) OSTERWEIL, L.J. and FOSDICK, L.D., "DAVE - A Validation, Error Detection, and Documentation System for Fortran Programs," Software- Practice and Experience,6, pp. 473-486, September 1976.

(2) FOSDICK, L.D., and OSTERWEIL, L.J., "Data Flow Analysis in Software Reliability," ACM Computing Surveys,8, pp. 305-330, September 1976.

(3) HUANG, J.C., "Detection of Data Flow Anomaly Through Program Instrumentation", IEEE Transactions on Software Engineering,Vol. SE-5,No. 3, May 1979.

4.12.1. Name. Execution Time Estimators/Analyzers.

4.12.2. Basic features. Execution time estimators/analyzers are tools which provide information about the execution characteristics of a program. They can be considered as validation tools in that they can be used to validate performance requirements and are part of the programming phase of the lifecycle.

4.12.3. Information input. The programs which are to have their execution performance monitored are, essentially, the input needed by the tool. Depending on the sophistication of the particular tool being used, the programs may be processed by a processor which automatically inserts probes to measure performance or probes may need to be manually inserted. The probes usually consist of calls to a monitor which records execution information such as CPU and I/O time, and statement execution counts.

4.12.4. Information outputs. The output produced by execution time estimators/analyzers are reports which show either by statement and/or module the execution time distribution characteristics. For example, a tool will provide information showing per module the number of entries to the module, cumulative execution time, mean execution time per entry and the percent execution time of the module with respect to the total program execution time.

4.12.5. Outline of method. Execution time estimators and execution time analyzers both perform similar functions but in different ways. Execution time estimators (1) function much in the same way as test coverage analyzers. A source program is instrumented with probes which collect statement execution counts when executed. Associated with each statement is a machine dependent estimate of the time required to execute the statement. The execution time estimate is multiplied by the statement execution count to give an estimate of the total time spent executing the statement. This is done for all statements in a program. Reports showing execution time breakdowns by statement, module, statement type, etc. can be produced.

Execution time analyzers are not usually as sophisticated as execution time estimators. Probes to measure the actual execution time of modules or program segments are inserted (usually by nand) into the source program. When the program has completed its execution, but just before it terminates, a routine is called which prints a report showing the execution characteristics of the monitored portions of the program.

The value of the tool lies primarily in its use as a performance requirements validation tool. In order to be used to formally validate performance requirements, however, it is necessary for the performance requirements to have been clearly stated and associated with specific functional requirements. Moreover, the system should have been designed so that the functional requirements can be traced to specific system modules.

Assuming that the above conditions are met, the tool could be used in the following way. The program to be analyzed would be monitored by the execution time estimator/analyzer during testing. The execution times for the modules corresponding to specific functional requirements would be compared with the

performance requirement for that function. Those modules which fail to satisfy their performance requirements would be studied in more detail for possible efficiency improvements. The tool results can also help to identify execution time critical sections of code. Once the necessary optimitions have been made, the program should be again tested using the tool to validate the performance requirements.

4.12.6. Example.

a. Application. A particular module in a real time, embedded computer system is required to perform its function within a specific time period. If not, a critical time dependent activity cannot be performed, resulting in the loss of the entire system.

b. Error. The module in question contained an error which involved performing unnecessary comparisons during a table look-up function although the proper table entry was always found.

c. Error detection. The problem was discovered during system testing using an execution time analyzer which clearly indicated that the offending module was not able to meet its performance requirements. The specific error was discovered on further examination of the module.

4.12.7. Effectiveness. The use of execution time estimators/analyzers (as well as test coverage analyzers) has uncovered an interesting property of many programs. The majority of the execution time spent by a program is spent executing a very small percentage of the code. Knowledge gained of where this execution time critical code is located through the use of an execution time estimator/analyzer can be extremely helpful in optimizing a program in order to satisfy performance requirements and/or reduce costs.

4.12.8. Applicability. Execution time estimators/analyzers can be used in any application.

4.12.9. Learning. The learning required is simply that which is necessary to execute the tool.

4.12.10. Costs. The tool is automated and therefore does involve some cost. The amount will depend on the tool's sophistication, but generally will not be excessive.

4.12.11. References.

(1) "PPE Users Guide", Boole and Babbage, No. U-D503-0.

(2) "Poseidon MK 88 Fire Control System Computer Program Verification and Validation Techniques Study", Vol. III, Ultrasystems, Inc., 500 Newport Center Dr., Newport Beach, CA, Nov. 1973.

4.13.1. Name. Formal Reviews.

4.13.2. Basic features. Formal reviews constitute a series of reviews of a system, usually conducted at major milestones in the system development lifecycle. They are used to improve development visibility and product quality and provide the basic means of communication between the project team, company management, and user representatives. They must provide judgmental decisions made by a team of blue ribbon specialists with a proven knowledge of current system operations. Formal reviews are most often implemented for medium to large size development projects, although small projects often employ a less rigorous form of the technique.

The most common types of formal reviews are held at the completion of the Requirements, Preliminary Design, Detailed (Critical) Design, Coding, and Installation phases. Whereas names of these reviews may vary by company, some generally recognized names are: Requirements Review, Preliminary Design Review (PDR), Critical Design Review (CDR), Code Construction Review, and Acceptance Test Review.

4.13.3. Information input. The input to a particular formal review will vary slightly depending on the stage of the lifecycle just completed. In general, each formal review will require that some sort of review package be assembled and then distributed at a review meeting. This package commonly contains a summary of the requirements which are the basis for the product being reviewed. These and other common inputs to formal reviews fall into three main categories, described below.

a. Project documents. These are documents produced by the development team to describe the system. The specific documents required are dependent upon the lifecycle phase just completed. For example: a review conducted at the conclusion of the requirements phase would necessitate availability of Functional Specifications or System/Subsystem Specifications.

b. Backup documentation. This type of input is documentation which is not usually contractually required, yet preparation of which is necessary to support systems development or otherwise record project progress. Specific types of backup documentation vary by the phase for which the review is performed. Rough drafts of user and maintenance manuals are examples of backup documentation examined during a design review to plan for continuation of the project. Program listings are an example of backup documentation utilized during a code construction review.

c. Other inputs. All other inputs are primarily used to clarify or expand upon the project documents and backup documents. They may include viewfoils and slides prepared by project management for the formal review meeting, the minutes of the previous phase review meeting, or preliminary evaluations of the project documents under review.

4.13.4. Information output. The information output associated with a formal review generally falls into the following categories.

a. Management reports. These are written reports from the project manager to upper management describing the results of the review, problems revealed, proposed solutions, and any upper management assistance required.

b. Outside reviewer reports. These are written reports to the project manager from participants of the review who have not worked on the project. These reports provide outside reviewers an opportunity to express their appraisal of the project status and the likelihood of meeting project objectives. It also allows them to make suggestions for correcting any deficiencies noted.

c. Action items. This is a list of all required post-review action items to be completed before a review can be satisfactorily closed out. With each item is an indication of whether customer or contractor action is required for resolution.

d. Review minutes. This is a written record of the review meeting proceedings which are recorded by a designee of the leader of the review team. The minutes of the review are distributed to each review team member after the completion of the review meeting.

e. Decision to authorize next phase. A decision must be reached at any formal review to authorize initiation of the next lifecycle phase.

f. Understanding of project status. At the conclusion of any formal review there should be a common understanding of project status among the project personnel present at the review.

4.13.5. Outline of method.

a. Participants. The participants in a formal review are often selected from the following group of people:

- o Project manager
- o Project technical lead
- o Other project team members - analysts, designers, programmers
- o Client
- o User representative(s)
- o Line management of project manager
- o Outside reviewers - quality assurance personnel, experienced people on other projects
- o Functional support personnel - finance, technology personnel
- o Subcontractor management, if applicable
- o Others - configuration management representative, maintenance representative

b. The process. Formal reviews should be scheduled and organized by project management. Each review must be scheduled at a meaningful point during system development. The review effectively serves as the phase milestone for any particular phase.

There are five basic steps involved in every formal review.

1. Preparation. All documentation that serves as source material for the review must be prepared prior to the meeting. These materials may be distributed to each participant before the meeting in order to allow sufficient time to review and make appraisals of the materials. The location and time of the meeting must be established, participants must be identified, and an agenda planned.

2. Overview presentation. At the review meeting, all applicable Product and Backup Documentation is distributed and a high-level summary of the product is presented. Objectives are also given.

3. Detailed presentation. A detailed description of the project status and progress achieved during the review period is presented. Problems are identified and openly discussed by the team members.

4. Summary. A summary of the results of the review is given. A decision about the status of the product is made and a list of new action items is constructed and responsibility for completion of each item is assigned.

5. Follow-up. The completion of all action items is verified. All reports are completed and distributed.

4.13.6. Example. By contract agreement, two weeks prior to completion of the requirements document, the producer of a program receives notification from his client that a requirements review meeting is desired. The client notifies a preselected chairperson to conduct the meeting. For participants the chairperson has selected the project manager, project technical lead, a member of the requirements definition team, and a member of the requirements analysis team. The client also has indicated that he would like to include the following people in the review: a representative from the user shop, a reviewer from an independent computing organization, and a representative from his own organization.

The chairperson informs all review participants of the date, time, and location of the review. Ten days prior to the meeting, the chairperson distributes all documents produced by the requirements definition and analysis teams (requirements document, preliminary plans, other review material) to each participant. In preparation of the meeting, each reviewer critically inspects the documents. The user representative is puzzled over the inclusion of a requirement concerning the use of a proposed database. The reviewer from the outside computing organization notes that the version of the operating system to be used in developing the system is very outdated. A representative of the client organization has a question concerning the use of a subcontractor in one phase of the project. Each reviewer submits his comments to the chairperson before the scheduled review meeting. The chairperson receives the comments and directs each to the appropriate requirements team member to allow proper time for responses to be prepared.

The requirements review meeting begins with a brief introduction by the chairperson. All participants are introduced, review materials are listed, and the procedure for conducting the review is presented. A presentation is then given summarizing the problem that led to the requirements and the procedure that was used to define these requirements. At this time, the user representative inquires about the requirement concerning the use of a particular database as stated in the requirements document. The project technical lead responds to this question. The user representative accepts this response, which is so noted by the recorder in the official minutes of the meeting.

The meeting continues with an analysis of the requirements and a description of the contractor's planned approach for developing a solution to the problem. At this time, the questions from the client representative and the outside computing organization are discussed. The project manager responds to questions concerning the use of a subcontractor on the project. Certain suggestions have been made which require the approval of the subcontractor. These suggestions are placed on the action list. The technical lead acknowledges the problems that the independent computing organization has pointed out. He notes that certain system vendors must be contacted to resolve the problem. This item is also placed on the action list. A general discussion among all review team members follows.

At the end of the review, the chairperson seeks a decision from the reviewers about the acceptability of the requirements document. They agree to give their approval, providing that the suggestions noted on the action list are thoroughly investigated. All participants agree to this decision and the meeting is adjourned.

The chairperson distributes a copy of the minutes of the meeting, including action items, to all participants. The project manager informs the subcontractor of the suggestions made at the meeting. The subcontractor subsequently agrees with the suggestions. The project technical leader contacts the system vendor from which the current operating system was purchased and learns that the latest version can be easily installed before it is needed for this project. He notifies the project manager of this, who subsequently approves its purchase. The requirements document is appropriately revised to reflect the completion of these action items. The chairperson verifies that all action items have been completed. The project manager submits a Management Report to management, summarizing the review.

4.13.7. Effectiveness. Since the cost to correct an error increases rapidly as the development process progresses, detection of errors by the use of formal reviews is an attractive prospect.

Some of the qualitative benefits attributable to the use of formal reviews are given below:

- o Highly visible systems development
- o Early detection of design and analysis errors
- o More reliable estimating and scheduling
- o Increased product reliability, maintainability

- o Increased education and experience of all individuals involved in the process
- o Increased adherence to standards
- o Increased user satisfaction

Little data is available which identifies the quantitative benefits attributable to the use of formal reviews.

Experience with this technique indicates that it is most effective on large projects. The costs involved in performing formal reviews on small projects, however, may be sufficiently large enough to consider lessening the formality of the reviews or even eliminating or combining some of them.

4.13.8. Applicability. Formal reviews are applicable to large or small projects following all development phases and are not limited by project type or complexity.

4.13.9. Learning. This technique does not require any special training. However, the success or failure of a formal review is dependent on the people who attend. They must be intelligent, skilled, knowledgeable in a specific problem area, and be able to interact effectively with other team members. The experience and expertise of the individual responsible for directing the review is also critical to the success of the effort.

4.13.10. Costs. The method requires no special tools or equipment. The main cost involved is that of human resources. If formal reviews are conducted in accordance with the resource guidelines expressed in most references, the cost of reviews for average programs are not high. However, the cost of reviewing major programs can be significant. Most references suggest that formal review meetings should not require more than 1 to 2 hours. Preparation time can amount to as little as 1/2 hour and should not require longer than 1/2 day per review.

4.13.11. References.

(1) FREEDMAN, D.P., and WEINBERG, G.M., "Ethno - Technical Review Handbook", 1977 Ethnotech,Inc..

(2) WEINBERG, G.M., "Programming as a Social Activity," The Psychology of Computer Programming,Van Nostrand, Reinhold, 1971.

(3) MYERS, G., "Reliable Software Through Composite Design", Petrocelli/Charter, 1975.

(4) SHNEIDERMAN, BEN, "Software Psychology - Human Factors in Computer and Information Systems", Winthrop Publishing, 1980.

(5) GLASS, R., "Software Reliability Guidebook", Prentice-Hall, Englewood Cliffs, N.J., 1979.

4.14.1. Name. Formal Verification.

4.14.2. Basic features. The purpose of formal verification is to apply the formality and rigor of mathematics to the task of proving the consistency between an algorithmic solution and a rigorous, complete specification of the intent of the solution.

4.14.3. Information input. The two inputs which are required are the solution specification and the intent specification. The solution specification is in algorithmic form, often but not always, executable code. The intent specification is descriptive in form, invariably consisting of assertions, usually expressed in Predicate Calculus.

Additional inputs may be required depending upon the rigor and specific mechanisms to be employed in the consistency proof. For example, the semantics of the language used to express the solution specification are required and must be supplied to a degree of rigor consistent with the rigor of the proof being attempted. Similarly, simplification rules and rules of inference may be required as input if the proof process is to be completely rigorous.

4.14.4. Information output. The proof process may terminate with a successfully completed proof of consistency, or a demonstration of inconsistency, or it may terminate inconclusively. In the former two cases, the proofs themselves and the proven conclusion are the outputs. In the latter case, any fragmentary chains of successfully proven reasoning are the only meaningful output. Their significance is, as expected, highly variable.

4.14.5. Outline of method. The usual method used in carrying out formal verification is Floyd's Method of Inductive Assertions or a variant thereof. This method entails the partitioning of the solution specification into algorithmically straightline fragments by means of strategically placed assertions. This partitioning reduces the proof of consistency to the proof of a set of smaller, generally much more manageable lemmas.

Floyd's Method dictates that the intent of the solution specification be captured by two assertions. The first assertion is the input assertion which describes the assumptions about the input. The second assertion is the output assertion which describes the transformation of the input, which is intended to be the result of the execution, of the specified solution. In addition, intermediate assertions must be fashioned and placed within the body of the solution specification in such a way that every loop in the solution specification contains at least one intermediate assertion. Each such intermediate assertion must express completely the transformations which are intended to occur or are occurring at the point of placement of the assertion.

The purpose of placing the assertions as just described is to assure that every possible program execution is decomposable into a sequence of straightline algorithmic specifications, each of which is bounded on either end by an assertion. If it is known that each terminating assertion is necessarily implied by executing the specified algorithm under the conditions of the initial assertion, then, by induction, it can be shown that the entire

execution behaves as specified by the input/output assertions, and hence as intended. For the user to be assured of this, Floyd's Method directs that a set of lemmas be proven. This set consists of one lemma for each pair of assertions which is separated by a straightline algorithmic specification and no other intervening assertion. For such an assertion pair, the lemma states that, under the assumed conditions of the initial assertion, execution of the algorithm specified by the intervening code necessarily implied the conditions of the terminating assertion. Proving all such lemmas establishes what is known as "partial correctness." Partial correctness establishes that whenever the specified solution process terminates, it has behaved as intended. In addition, total correctness is established by proving that the specified solution process must always terminate. This is clearly an undecidable question, being equivalent to the Halting Problem, and hence its resolution is invariably approached through the application of heuristics.

In the above procedure, the pivotal capability is clearly the ability to prove the various specified lemmas. This can be done to varying degrees of rigor, resulting in proofs of corresponding varied degrees of reliability and trustworthiness. For the greatest degree of trustworthiness, solution specification, intent specification, and rules of reasoning must all be specified with complete rigor and precision. The principal difficulty here lies in specifying the solution with complete rigor and precision. This entails specifying the semantics of the specification language, and the functioning of any actual execution environment with complete rigor and precision. Such complete details are often difficult or impossible to adduce. They are, moreover, when available, generally quite voluminous, thereby occasioning the need to prove lemmas which are long and intricate.

4.14.6. Example. As an example of what is entailed in a rigorous formal verification activity, consider the specification of a bubble sort procedure. (The details of this can be found in Reference 3 for this technique.) The intent of the bubble sort must first be captured by an input/output assertion pair. Next, observing that the bubble solution algorithm contains two nested loops, leads to the conclusion that two additional intermediate assertions might be fashioned, or perhaps one particularly well placed assertion might suffice. In the former case, up to eight lemmas would then need to be established; one corresponding to each of the (possible two) paths from the initial to each intermediate assertion, one corresponding to each of the two paths from an intermediate assertion back to itself, one for each of the (possibly two) paths from one intermediate assertion to the other, and finally one for each of the (possibly two) paths from intermediate to terminating assertion. Each lemma would have to be established through rigorous mathematical logic (see Reference 3). Finally, a proof of necessary termination would need to be fashioned (see Reference 3).

4.14.7. Effectiveness. The effectiveness of formal verification has been attacked on several grounds. First and most fundamentally, formal verification can only establish consistency between intent and solution specification. Hence, inconsistency can indicate error in either or both. The same can be said for most other verification techniques, however. What makes this particularly damaging for formal verification is that complete rigor and detail in the intent specification are important, and this

requirement for great detail invites error.

The amount of detail also occasions the need for large, complex lemmas. These, especially when proven using complex, detailed rules of inference, produce very large, intricate proofs which are highly prone to error.

Finally, formal verification of actual programs is further complicated by the necessity to express rigorously the execution behavior of the actual computing environment for the program. As a consequence of this, the execution environment is generally modeled incompletely and imperfectly, thereby restricting the validity of the proofs in ways which are difficult to determine.

Despite these difficulties, a correctly proven set of lemmas establishing consistency between a complete specification and a solution specification whose semantics are accurately known and expressed conveys the greatest assurances of correctness obtainable. This ideal of assurance seems best attainable by applying automated theorem provers to design specifications, rather than code.

4.14.8. Applicability. Formal verification is a technique which can be applied to determine the consistency between any algorithmic solution specification and any intent specification. As elaborated upon earlier, however, the trustworthiness of the results is highly variable depending primarily upon the rigor with which the specifications are expressed and the proofs are carried out. Formal verification is best employed on critical code where errors have severe consequences.

4.14.9. Learning. As noted, the essence of this technique is mathematical. Thus, the more mathematical sophistication and expertise which practitioners possess, the better. In particular, a considerable amount of mathematical training and expertise is necessary for the results of applying this technique to be significantly reliable and trustworthy.

4.14.10. Costs. This technique, when seriously applied, must be expected to consume very significant amounts of the time and effort of highly trained mathematically proficient personnel. Hence, considerable human-labor expense must be expected.

As noted earlier, human effectiveness can be considerably improved through the use of automated tools such as theorem provers. It is important to observe, however, that such tools can be prodigious consumers of computer resources. Hence, their operational costs are also quite large.

4.14.11. References.

(1) FLOYD, R.W., "Assigning Meanings to Programs," in Mathematical Aspects of Computer Science, 19, SCHWARTZ, J.T. (ed.), American Mathematical Society, Providence, R.I., pp.19-32, 1967.

(2) ELSPAS, B., et al., "An Assessment of Techniques for Proving Program Correctness, " ACM Computing Surveys, 4, pp.97-147, June, 1972.

(3) GOOD, D.I., LONDON, R.L., and BLEDSOE, W.W., "An Interactive Program Verification System," Proceedings 1975 International Conference on Reliable Software, IEEE Catalog 75CH0940-7CSR, pp.482-492.

(4) HOARE, C.A.R., "An Axiomatic Basis for Computer Programming," CACM, 12, October 1969, pp. 576-583.

4.15.1. Name. Global Roundoff Analysis of Algebraic Processes

4.15.2. Basic features. The technique involves the use of computer software to locate numerical instabilities in algorithms consisting of algebraic processes. Global roundoff analysis is the determination of how rounding error propagates in a given numerical method for many or all permissible sets of data. This technique has two areas of application: Case I - to decide whether an algorithm is as accurate as can be expected given the fundamental limitation of finite precision arithmetic; and Case II - to decide which of two competing algorithms is "more stable," i.e., less susceptible to rounding errors.

4.15.3. Information input.

a. Case I - Analysis of a single algorithm
(i) algorithm described in a simple programming language
(ii) data set for algorithm
(iii) choice and type of rounding error measures
(iv) stopping value for maximizer

b. Case II - Comparison of two algorithms
(i) each algorithm described in a simple programming language
(ii) data set for algorithms
(iii) choice of rounding error measure and mode of comparison
(iv) stopping value for maximizer

4.15.4. Information output.

a. Case I - Analysis of a single algorithm
(i) output computed for the initial data set
(ii) list of values found by the maximizer
(iii) final set of data
(iv) if instability diagnosed, then all arithmetic operations at the final set of data are listed

b. Case II - Comparison of two algorithms
(i) output computed for the initial algorithms
(ii) list of values found by the maximizer
(iii) final set of data

4.15.5. Outline of method. For an algorithm and a data set, $\bar{d}$, then:

(a) A function $w(\bar{d})$, called a Wilkinson number, has been defined which measures the effects of rounding errors. Large values for w is the sign of an unstable algorithm.

(b) Wilkinson number has been shown to be a "smooth" function of $\bar{d}$, i.e. as the original data set values are altered in small increments, the values of w are correspondingly altered in small increments.

(c) An approximation to Wilkinson numbers has been developed which is straight forward to compute.

(d) The representation of the algorithm is analyzed.

(e) Using the initial data set as a starting point, the global analysis program uses numerical maximization techniques to modify the data set The search is directed toward finding a data set with a disastrously large value of $w(\bar{d})$.

4.15.6. Example. Triangular Matrix Inversion (4). The better matrix inversion algorithms are known from experience to almost invariably produce satisfactory results. However, the question remains whether there is a guarantee that the results are always good. The question can be reformulated as: Is the traditional back substitution algorithm for inverting an upper triangular matrix numerically stable in the sense that there is a modest bound, depending on the matrix size, for w? To apply the technique, the algorithm is represented as a program in figure 4.15.6-1. Note that the statement "TEST (N=4)" indicates that the search for numerical stability will be conducted in the domain of 4x4 matrices. An approximation to w, W_4, will be calculated.

```
      TEST (N=4)
C     COMPUTE S = (T INVERSE), WHERE T IS A NONSINGULAR, UPPER
C     TRIANGULAR MATRIX.
      DIMENSION (S(N,N),T(N,N))
C     INPUT T.
        FOR J = 1 to N BY 1
            FOR I = J to 1 BY -1
                INPUT (T(I,J))
                END (I)
            END(J)
C
C     COMPUTE S.
        FOR K = 1 TO N BY 1
            S(K,K) = 1.0/T(K,K)
            FOR I = K-1 TO 1 BY -1
                S(I,K) = -SUMMATION(T(I,J)*S(J,K),J=I+1 TO K)/T(I,I)
                END (I)
            END(K)
C
C     OUTPUT S.
        FOR J=1 TO N BY 1
            FOR I=J TO 1 BY -1
                OUTPUT(S(I,J))
                END(I)
            END(J)
        STOP
```

Figure 4.15.6-1 Triangular Matrix Inversion

The compiler portion of the package checks the program for errors, then translates them into a form suitable for analysis.

The initial data set for the search for numerical instability was:

$$T_0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ & 2 & 0 & 0 \\ & & 3 & 0 \\ & & & 4 \end{pmatrix}$$

The roundoff analysis program was told to seek a value of W in excess of 10,000. The maximizer located the following matrix:

$$T_\infty \approx \begin{pmatrix} -0.001 & 5.096 & 5.101 & 1.853 \\ & 3.737 & 3.740 & 3.392 \\ & & 0.0006 & 5.254 \\ & & & 4.567 \end{pmatrix}$$

with $W_4(T_\infty) > 10{,}000$ in 6 seconds CPU time on a IBM 370/168.

The fact that W_4 can be large for data like T seems implicit in known results, e.g., (6), verifying the ill behavior of triangular matrices with diagonal entries approaching zero.

4.15.7. Effectiveness. Failure of the maximizer to find large values of w does not guarantee that none exist (2). Thus, the technique tends to be optimistic; unstable methods may appear stable. However, experience indicates that this method is surprisingly reliable. At least, the failure of the maximizer to find large values of w can be interpreted as providing evidence for stability equivalent to a large amount of practical experience with low order matrices.

4.15.8. Applicability. The technique is intended for noniterative methods from numerical linear algebra.

4.15.9. Learning. Most algorithms should be able to be analyzed in 2 to 8 hours of training and preparation assuming the software is available.

4.15.10. Costs. The performance of the technique is related to the performance of the algorithm being checked.

4.15.11. References.

(1) MILLER, W., "Software for Roundoff Analysis", ACM Transactions on Mathematical Software, 1,2, June 1975, 108–128.

(2) MILLER, W., "Computer Search for Numerical Instability", Journal of the ACM,4,October 1975, 512-521.

(3) MILLER, W., "Roundoff Analysis by Direct Comparison of Two Algorithms", SIAM Journal of Numerical Analysis,13, 1976, 382-392.

(4) MILLER, W. and SPOONER, D., "Software for Roundoff Analysis, II", ACM Transactions on Mathematical Software4,4,1978, 369-387.

(5) MILLER, W. and SPOONER, D., "Algorithm 532 Software for Roundoff Analysis 2 ", ACM Transactions on Mathematical Software, 4,4,1978, 388-390.

(6) ANDERSON, A. and KARASALO, I., "On Computing Bounds for the Least Singular Value of a Triangular Matrix",BIT,1975,1-4.

4.16.1. Name. Inspection

4.16.2. Basic features. Informal reviews constitute a thorough inspection mechanism used to detect errors in system components and documentation. Several inspections are generally conducted for each item as it progresses through the lifecycle. The most commonly recognized inspections are conducted during the design and programming stages and are referred to as design inspections and code inspections. However, the inspection concept may be applied to any functionally complete part of a system during any or all phases of the lifecycle and are typified by utilization of checklists and summary reports. Another unique feature of an inspection is the use of data from past inspections to stimulate future detection of categories of errors.

4.16.3. Information input. The input required for each inspection falls into three main categories: relevant project documents, backup documentation, and inspection checklists.

a. Project documents. These are documents produced by the development team to describe the system. The specific documents required are dependent upon the lifecycle phase currently in progress. For example: an inspection conducted during the design phase would necessitate availability of Functional Specifications or System/Subsystem Specifications.

b. Backup documentation. This type of input is documentation which is not usually contractually required, yet preparation of which is necessary to support systems development or otherwise record project progress. Specific types of backup documentation vary by the phase in which the inspection is conducted. Data dictionaries and cross-reference tables are examples of backup documentation utilized during a design inspection. Program listings are an example of code inspection backup documentation.

c. Checklists. Each member of the inspection team uses a checklist for review preparation and during the course of the inspection itself. The checklist content may vary based upon the particular application being inspected and is updated from feedback of other recent inspections. For example, a checklist to be employed during a code inspection of a COBOL program component would contain items like:

- o Are specialized printer controls used to enhance component readability (e.g., use of EJECT or SKIP commands)?
- o Does each procedure have only one exit and entry?
- o Are IF-THEN-ELSE statements indented in a logical fashion?
- o Are file, record and data names representative of the information they contain and do they conform to established naming conventions?
- o Are comments explicit and accurate?
- .
- .
- .

etc.

4.16.4. Information output. The information output associated with an inspection is either related to inspection planning and scheduling or inspection results.

a. Inspection schedule memo. The memo is produced upon notification from management that an inspection should be forthcoming. The memo defines the roles and responsibilities of each inspection team member, estimated time required for each inspection task, and a summary of the status of the item being reviewed (including any previous inspections conducted).

b. Problem Definition Sheet/Error Description Summary. This form is used to record information about each detected error. It describes the location, nature, and classification of the errors.

c. Summary Report. A Summary Report is used to document correction of all errors reported during an inspection. Data recorded on the report is tabulated and becomes part of cumulative error statistics which can be used to improve the development and inspection processes.

d. Management Reports. These reports are the means by which management is informed about the types of errors being detected and the amount of resources being expended to correct them. The information from these reports highlights frequent sources of errors, providing input to management for future updates to the inspection checklist.

4.16.5. Outline of method.

a. Roles and Responsibilities. The group of people responsible for the inspection results are usually called an inspection team and are given responsibilities based upon their contribution to the item being inspected. The leader of the group is responsible for all process planning, moderating, reporting, and follow-up activities. The designer implementer (person responsible for building the item) and the tester of the item being inspected are also members of the inspection team. Management does not normally participate in an inspection.

b. The Process. There are five basic steps involved in every inspection: planning, preparation, inspection meeting, rework and follow-up. The first inspection for a particular item contains another step: overview presentation. These steps are summarized below.

While these steps should not vary functionally for inspections conducted at different development phases, the responsibilities of the individuals on the inspection team will necessarily vary slightly. This occurs because the primary responsibility for the item shifts as the lifecycle progresses. For example, during a design inspection, the designer is the focal point. However, during a code inspection or document inspection, the implementation is the focal point.

1. Planning. Set up inspection schedule and assemble inspection team.
2. Overview Presentation (conducted only for the first inspection of the item during the development process). Distribute applicable Product

and Backup Documentation and present a high level summary of the item to be inspected.

3. Preparation. Team members read and review documentation and list any questions.
4. Inspection Meeting. Conduct detailed description of the item, noting all errors detected. Use checklists to ensure inspection completeness and Problem Definition Report to summarize errors.
5. Rework. Estimate time to correct errors and implement the corrections.
6. Follow-up. Verify that all errors have been corrected using Problem Definition Sheet as a checklist. Complete Summary and Management Reports.

4.16.6. Example. The following is an example of a design inspection of a software component or item which defines the roles and responsibilities of the inspection team members. Upon decision of management to conduct a design inspection, the selected leader initiates process planning by identifying team members and their roles and responsibilities. If this is the first inspection for this item (i.e., there has been no requirements inspection), the leader next schedules an overview presentation. The project and backup documentation (i.e., Functions Specification, system flow charts, etc.) are distributed and the item designer leads the team through a high level description of the item.

After the presentation each team member reads and reviews the distributed documentation and lists any questions. This list of prepared questions is often given to the leader and/or designer prior to the inspection meeting.

At the designer inspection meeting the implementer leads the team through a detailed description of the design of the item being inspected. Backup documentation facilitates the description and clarifies points which may be brought up. The checklist is used by each team member to help identify errors and enforce standards. The problem definition sheet is prepared by the team leader at the end of the inspection. The item design will either be approved as-is, approved with modifications, or rejected. In the last two cases, the problem definition sheet is given to the designer and the correction process begins.

At the start of this rework process an estimate is made by the leader and designer specifying time required for correction. This estimate is entered on the Problem Definition Sheet and is provided to management. Management can then make a judgment as to whether their project schedule will be affected. Necessary changes to the item are made and the item is either reinspected or submitted to follow-up procedures.

During follow-up, the Problem Definition Sheet is used as a checklist for the leader and designer to verify that all errors have been analyzed and corrected. The reader then fills out the Summary and Management Reports and submits them to management.

4.16.7. Effectiveness. Since the cost to correct an error increases rapidly as the development process progresses, detection of errors by early use of inspections is an attractive prospect.

Studies have been carried out which indicate that inspections are an effective method of increasing product quality (reliability, usability and maintainability). Experience with the technique indicates that it is effective on projects of all sizes. The best results are generally achieved when the inspection leader is experienced in the inspection process.

Some of the best quantitative results of the use of inspections have come from IBM, which has been studying the use of the technique for a number of years. One study, detailing and comparing the benefits of inspections and structured walkthroughs, indicated 23% higher programmer productivity with inspections than with walkthroughs. No data was available documenting the amount of increased programmer productivity attributable to inspections alone. The study also reported 38% fewer errors in the running code than if solely applying walkthroughs as an error detection mechanism.

The qualitative benefits attributable to the use of inspections are substantative. The following list is illustrative of some of these positive effects:

- o Programs which are less complex
- o Subprograms which are written in a consistent style, complying with established standards
- o Highly visible systems development
- o More reliable estimating and scheduling
- o Increased education and experience of all individuals involved in the inspection process
- o Increased user satisfaction
- o Improved documentation
- o Less dependence on key personnel for critical skills

4.16.8. Applicability. While the most commonly used inspections are for design and code, the technique is not limited to these phases and can be applied during all phases, for most types of applications (i.e., business, scientific, etc.) on large or small projects.

4.16.9. Learning. The experience of the inspection leader is essential to the success of the effort. A correct attitude about the process is essential to all involved, including the appropriate managers. Many excellent texts about inspections (and other types of reviews) are in existence which should supply the required level of detail as well as discuss some team psychology issues pertinent to inspection conduct.

4.16.10. Costs. The method requires no special tools or equipment. The main cost involved is that of human resources. If inspections are conducted in accordance with the resource guidelines expressed in most references, the costs of inspections are negligible compared with the expected returns. It should be kept in mind that follow-up inspections to correct previously detected errors can increase the original cost estimation. Most references suggest that inspection meetings should last no longer than 2 hours, and can reasonably be kept to 15 minutes. Preparation time can amount to as little as 1/2 hour and should not require longer than 1/2 day per inspection.

4.16.11. References.

(1) "Code Reading: Structured Walkthroughs and Inspections", IBM, IPTO Support Group, World Trade Center, Postbus 60, Zoetermeer, Netherlands, March 1976.

(2) FAGEN, M.E., "Design and Code Inspections to reduce errors in Program Development", IBM Systems Journal, No. 3,1976.

(3) FREEDMAN, D.P. and WEINBERG, G.M., "Ethno - Technical Review Handbook", Ethnotech, Inc.,1977.

(4) "Systematic Software Development and Maintenance (SSDM)", BCS Document #10155,February 1977.

4.17.1. Name. Interactive Test Aids.

4.17.2. Basic features. Interactive test aids,debuggers, are tools used to control and/or analyze the dynamics of a program during execution. The capabilities provided by these tools are used to assist in identifying and isolating program errors. These capabilities allow the user to:

- o suspend program execution at any point to examine program status,
- o interactively dump the values of selected variables and memory locations,
- o modify the computation state of an executing program,
- o trace the control flow of an executing program.

4.17.3. Information input. Interactive test aids require as input the source code that is to be executed and the commands that indicate which testing operations are to be performed by the tool during execution. Included in the commands are indications of which program statements are to be affected by the tool's operation. Commands can be inserted in the source code and/or entered interactively by the user during program execution at preselected break points.

4.17.4. Information output. The information output by an interactive test aid is a display of requested information during the execution of a program. This information may include the contents of selected storage cells at specific execution points or a display of control flow during execution.

4.17.5. Outline of method. The functions performed by an interactive test aid are determined by the commands input to it. Some common commands are described below.

BREAK: Suspend program execution when a particular statement is executed or a particular variable is altered.

DUMP: Display the contents of specific storage cells, e.g., variables, internal registers, other memory locations.

TRACE: Display control flow during program execution through printed traces of:
- o statement executions (using statement labels or line numbers),
- o subroutine calls, or
- o alterations of a specified variable.

SET: Set the value of a specified variable.

CONTENTS: Display the contents of certain variables at the execution of a specific statement.

SAVE: Save the present state of execution.

RESTORE: Restore execution to a previously SAVEd state.

CALL: Invoke a subroutine.

EXECUTE: Resume program execution at a BREAK point.

EXIT: Terminate processing.

These commands allow complete user control over the computation state of an executing program. It allows the tester to inspect or change the value of any variable at any point during execution.

The capabilities of special interactive testing aids can also be found in many implementations of interpreters and compilers for such languages as BASIC, FORTRAN, COBOL, and PL/I.

4.17.6. Example. A critical section of code within a routine is to be tested. The code computes the values of three variables, X, Y, and Z, which later serve as inputs to other routines. To ensure that the values assigned to X, Y, and Z have been correctly computed in this section of code, an interactive testing aid is used to test the code.

Two BREAK commands are initially inserted into the code. A BREAK command is inserted immediately before the first statement and immediately after the last statement of the section of code being tested. To display the value of X, Y, and Z, a CONTENTS command is placed before the second BREAK command. The program containing the above mentioned code is executed. When the first BREAK command is encountered, execution is halted and a prompt is issued to the user requesting that a command be entered. A SAVE command is typed by the user in order to save the present state of execution. Then SET command is entered to set the values of two variables, A and B, which are used to compute the values of X, Y, and Z. The EXECUTE command is then issued to resume program execution.

At the end of execution of the relevant section of code the preinserted CONTENTS command displays the computed values of X, Y, and Z. The second BREAK command allows time for these values to be examined and gives the user the opportunity to enter new commands. At this time, a RESTORE command is entered that will restore the computation state to the state that was previously saved by the SAVE command. For this example, the computation state returns to that which followed the first BREAK command, allowing the code under analysis to be tested with different input values. Different values for A and B are entered and the contents of X, Y, and Z are observed as before. This process is repeated several times using carefully selected values for A and B and the corresponding values of X, Y, and Z are closely examined each time. If results of several computations look suspicious, their input and output values are noted and the code is more thoroughly examined. The program is finally terminated by entering the EXIT command at one of the two possible break points.

4.17.7. Effectiveness. To be an effective testing tool, an interactive test aid should be used with a disciplined strategy to guide the testing process. The tools can be easily misused if no testing methodology is combined with their use.

4.17.8. Applicability. Interactive test aids can be applied to any type of source code. Most existing tools, however, are language dependent (i.e., will operate correctly only for specified languages).

4.17.9. Learning. A minimal amount of learning is required to use these tools. It is comparable to the learning required in using a text editor. However, if the tool is to be used most efficiently, some learning is required in utilizing the tool with an effective testing strategy.

4.17.10. Costs. Programs executing under an interactive test aid will require more computing resources (e.g., execution time, memory for diagnostic tables) than if executed under normal operation. The cost is dependent on the implementation of the tool. For example, those based on interpretive execution will involve costs different from those driven by monitor calls.

4.17.11. References.

(1) MYERS, Glenford, "The Art of Software Testing," Wiley-Interscience, New York, 1975.

(2) "Sperry Univac Series 1100 FORTRAN (ASCII) Programmer Reference," Sperry Rand Corporation,1979.

(3) TAYLOR, R.N., MERILATT, R.L., and OSTERWEIL, L.J., "Integrated Testing and Verification System for Research Flight Software - Design Document," NASA CR 159095,July 31, 1979.

4.18.1. Name: Interface Checker.

4.18.2. Basic features. Interface checkers analyze the consistency and completeness of the information and control flow between components, modules or procedures of a system.

4.18.3. Information input. Information needed by interface checkers consists of either:

a. a formal representation of system requirements or
b. a formal representation of system design or
c. a program coded in a high-level language.

4.18.4. Information output. Module interface inconsistencies and errors are revealed. The information can be provided as error messages included with a source listing or as a separate report.

4.18.5. Outline of method. Interface checkers are fully automated tools which analyze a computer processable form of a software system requirements specification, design specification or code. The method for each of the three representations -- requirements, design, and code -- will be illustrated below by examining the interface checking capabilities of three existing tools.

PSL/PSA (Problem Statement Language/Problem Statement Analyzer) (1) is an automated requirements specification tool. Basically, PSL/PSA describes system requirements as a system of inputs, processes and outputs. Both information and control flow are represented within PSL. Interface checking performed by PSA consists of ensuring that all data items are used and generated by some process and that all processes use data. Incomplete requirements specification are, therefore, easily detected.

The Design Assertion Consistency Checker (DACC) (2) is a tool which analyzes module interfaces based on a design which contains information describing, for each module, the nature of the inputs and outputs. This information is specified using assertions to indicate the number and order of inputs, data types, units (e.g., feet or radians), acceptable ranges, and so on. DACC checks module calls against the assertions in the called module for consistency. This produces a consistency report indicating which assertions have been violated.

PFORT (3) is a static analysis tool which is primarily used for checking Fortran programs for adherence to a portable subset of the Fortran language but it also performs subprogram interface checking. PFORT matches actual with dummy arguments and checks for unsafe references, such as constraints being passed as arguments.

Interface checking capabilities can also be included within a particular language's compiler as well. For example, Ada (4) provides a parameter passing mechanism whereby parameters are identified to be input or output or input/output. Moreover, data type and constraints (e.g., range and precision) must match between the actual arguments and the formal parameters (in non-generic subprograms).

In summary, interface checking tools will generally check for:

- o modules which are used but not defined,
- o modules which are defined but not used,
- o incorrect number of arguments,
- o data type mismatches between actual and formal parameters,
- o data constraint mismatches between actual and formal parameters,
- o data usage anomalies.

4.18.6. Example.

a. Application. A statistical analysis package written in Fortran utilizes a file access system to retrieve records containing data used in the analysis.

b. Error. The primary record retrieval subroutine is always passed a statement number in the calling program which is to receive control in case an abnormal file processing error occurs. This is the last argument in the argument list of the subroutine call. One program, however, fails to supply the needed argument. The compiler is not able to detect the error. Moreover, the particular Fortran implementation is such that no execution time error occurs until a return to the unspecified statement number is attempted, at which time the system crashes.

c. Error discovery. This error can easily be detected by using an interface checker at either the design (e.g., DACC) or coding phase (e.g., PFORT) of the software development activity. Both DACC and PFORT can detect incorrect numbers of arguments.

4.18.7. Effectiveness. Interface checkers are very effective at detecting a class of errors which can be difficult to isolate if left to testing. They are generally more cost effective if provided as a capability within another tool such as a compiler, data flow analyzer or a requirements/design specification tool.

4.18.8. Applicability. The method is generally applicable.

4.18.9. Learning. The use of interface checkers requires only a very minimal learning effort.

4.18.10. Costs. Interface checkers are quite inexpensive to use, usually much less than the cost of a compilation.

4.18.11. References.

(1) TEICHROW, D. and HERSHEY III, E.A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering, SE-3, 1977(41-48).

(2) BOEHM, B., McCLEAN, R. and URFRIG, D., "Some Experience with Automated Aides to the Design of Large-scale Reliable Software", IEEE Transactions on Software Engineering, SE-1, 1975 (125-133).

(3) RYDER, B.G. and HALL, A.D., "The PFORT Verifier", Computing Science Technical Report#12, Bell Labs, March 1975.

(4) "Preliminary Ada Reference Manual", SIGPLAN Notices, Vol. 14, No. 6, part A, (June, 1979).

4.19.1. Name. Mutation analysis.

4.19.2. Basic features. Mutation analysis is a technique for detecting errors in a program and for determining the thoroughness with which the program has been tested. It entails studying the behavior of a large collection of programs which have been systematically derived from the original program.

4.19.3. Information inputs. The basic input required by mutation analysis is the original source program and a collection of test data sets on which the program operates correctly, and which the user considers to adequately and thoroughly test the program.

4.19.4. Information outputs. The ultimate output of mutation analysis is a collection of test data sets and good assurance that the collection is in fact adequate to thoroughly test the program. It is important to understand that the mutation analysis process may very well have arrived at this final state only after having exposed program errors and inadequacies in the original test data set collection. Hence, it is not unreasonable to consider errors detected, new program understanding, and additional test data sets to also be information outputs of the mutation analysis process.

4.19.5. Outline of method. The essential approach taken in the mutation analysis of a program is to produce from the program a large set of versions, each derived from a trivial transformation of the original, and to subject each version to testing by the given collection of test data sets. Because of the nature of the transformations, it is expected that the derived versions will be essentially different programs from the original. Thus, the testing regimen should demonstrate that each is in fact different. Failure to to so invites suspicion that the collection of test data sets is inadequate. This usually leads to greater understanding of the program and either the detection of errors or an improved collection of test data sets, or both.

A central feature of mutation analysis is the mechanism for creating the program mutations -- the derived versions of the original program. The set of mutations which is generated and tested is the set of all programs which differ from the original only in a small number (generally 1 or 2) of textual details, such as a change in an operator, variable or constant. Research appears to indicate that larger numbers of changes contribute little or no additional diagnostic power.

The basis for this procedure is the "Competent Programmer" assumptions which state that program errors are not random phenomena, but rather result from lapses of human memory or concentration. Thus, an erroneous program should be expected to differ from the correct one only in a small number of details. Hence, if the original program is incorrect, then the set of all programs created by making a small number of the small textual changes just described should include the correct program. A thorough collection of test data sets would reveal behavioral differences between the original, incorrect program and the derived correct one.

Hence, mutation analysis entails determining whether each mutant behaves differently from the original. If so, the mutant is considered incorrect. If not, the mutant must be studied carefully. It is entirely possible that the mutant is in fact functionally equivalent to the original program. If so, its identical behavior is clearly benign. If not, the mutant is highly significant, as it certainly indicates an inadequacy in the collection of test data sets. It may, furthermore, indicate an error in the original program which previously went undetected because of inadequate testing. Mutation analysis facilitates the detection of such errors by automatically raising the probability of each such error and then demanding justification for concluding that each has not in fact been committed. Most mutations quickly manifest different behavior under exposure to any reasonable test data set collection, and thereby demonstrate the absence of the error corresponding to the mutation by which they were created. This forces detailed attention on those mutants which behave identically to the original and thus forces attention on any actual errors.

If all mutations of the original program reveal different execution behavior, then the program is considered to be adequately tested and correct within the limits of the "Competent Programmer" assumption.

4.19.6. Example. Consider the Fortran program, figure 4.19.6-1, which counts the number of negative and non-negative numbers in array A:

```
      SUBROUTINE COUNT (A, NEG, NONNEG)
      DIMENSION A(5)
      NEG=0
      NONNEG=0
      DO 10 I=1,5

      IF (A(I).GT.0) NONNEG=NONNEG+1
      IF (A(I).LT.0) NEG=NEG+1

10    CONTINUE
      RETURN
      END
```

Figure 4.19.6-1 Subroutine Count

and the collection of test data sets produced by initializing A in turn to:

I	II	III
1	1	-1
-2	2	-2
3	3	-3
-4	4	-4
5	5	-5

Mutants might be produced based upon the following alterations:
a. Change an occurrence of any variable to any other variable, e.g.,

A to I
NONNEG to NEG
I to NEG
.
.
.

b. Change an occurrence of a constant to another constant which is close in value:
e.g.,
1 to 0
0 to 1
0 to -1
1 to 2

c. Change an occurrence of an operator to another operator:
e.g.,
NEG + 1 to NEG * 1
NEG + 1 to NEG - 1
A(I).GT.0 to A(I).GE.0
A(I).LT.0 to A(I).NE.0

Thus, the set of all "single alteration" mutants would consist of all programs containing exactly one of the above changes. The set of all "double alteration" mutants would consist of all programs containing a pair of the above changes.

Clearly many such mutations are radically different and would quickly manifest obviously different behavior. For example, in changing variable I to A (or vice versa) the program is rendered uncompilable by most compilers. Similarly changing "NEG=0" to "NEG=1" causes a different outcome for test case I.

Significantly, changing A(I)GT.0 to A(I).GE.0 or A(I).LT.0 to A(I).LE.0 produces no difference in run-time behavior on any of the three test data sets. This rivets attention on these mutants, and subsequently on the issue of how to count zero entries. One rapidly realizes that the collection of test data sets was inadequate in that it did not include any zero input values. Had it included one, it would have indicated that:

IF (A(I).GT.0) NONNEG=NONNEG+1 should have been
IF (A(I).GE.0) NONNEG=NONNEG+1.

Thus, mutation analysis has pointed out both this error and this weakness in the collection of test data sets. After changing the program and collection, all mutants will behave differently strongly raising our confidence in the correctness of the program.

4.19.7. Effectiveness. Mutation analysis can be an effective technique for detecting errors, but it must be understood that it requires combining an insightful human with good automated tools. Even then it must be understood that it is a reliable technique for demonstrating the absence only if all possible mutation errors (i.e., those involving alteration, interchanging, or

omission of operators, variables, etc.) are examined.

The need for good tools is easily understood when one realizes that any program has an enormous number of mutations, each of which must be generated, exercised by the test data sets, and evaluated. On the surface, this would appear to entail thousands of edit runs, compilations and executions. Clever tools have been built, however, which operate off a special internal representation of the original program. This representation is readily and efficiently transformed into the various mutations, and also serves as the basis for very rapid simulation of the mutants' executions, thereby avoiding the need for compilation and loading of each mutant.

This tool set still does not bypass the need for humans, however. Humans must still carry out the job of scrutinizing mutants which behave identically to the original program in order to determine whether the mutant is equivalent or whether the collection of test data sets is inadequate.

At the end of a successful mutation analysis, many errors may have been uncovered, and the collection of test data sets has certainly been made very thorough. Whether the absence of errors has been established, however, must be considered relative to the "Competent Programmer" assumption. Under this assumption, clearly all errors of mutation are detectable by mutation analysis; thus, the absence of diagnostic messages or findings indicates the absence of these errors. Mutation analysis cannot, however, assure the absence of errors which cannot be modeled as mutations.

4.19.8. Applicability. Mutation analysis is apparently applicable to any algorithmic solution specification. As previously indicated, it can only be considered effective when supported by a body of sophisticated tools. Tools enabling analysis of Fortran and COBOL source text exist. There is, furthermore, no reason why tools for other coding languages, as well as algorithmic design languages, could not be built.

4.19.9. Learning. This technique requires the potential mutation analyst to become familiar with the philosophy and goals of this novel approach. In addition it appears that the more familiar the analyst is with the subject algorithmic solution specifications, the more effective the analyst will be. This is because the analyst may well have to analyze a collection of test data sets to determine how to augment it, and may have to analyze two programs to determine whether they are equivalent.

4.19.10. Costs. In view of the previous discussion, it is important to recognize that significant amounts of human analyst time are likely to be necessary to do mutation analysis. The computer time required is not likely to be excessive if the sophisticated tools described earlier are available. The interested reader is urged to consult the following references for explanation of this.

4.19.11. References.

(1) DEMILLO, R.A., LIPTON, R.J. and SAYWARD, F.G., "Program Mutation: A New Approach to Program Testing", Infotech State-of-the-Art Report on Software Testing, V.2, INFOTECH/SRA, 1979, pp. 107-127.

(2) LIPTON, R.J. and SAYWARD, F.G., "The Status of Research on Program Mutation", Digest of the Workshop on Software Testing and Test Documentation, Fort Lauderdale, Fla. 1978, pp. 355-373.

4.20.1. Name. Peer Review

4.20.2. Basic Features. A peer review is a process by which project personnel perform a detailed study and evaluation of code, documentation, or specification. The term peer review refers to product evaluations which are conducted by individuals of equal rank, responsibility, or of similar experience and skill. There are a number of review techniques which fall into the overall category of a peer review. Code reading, round-robin reviews, walkthroughs and inspections are examples of peer reviews which differ in formality, participant roles and responsibilities, output produced and input required.

4.20.3. Information input. The input to a particular peer review will vary slightly depending on which form of peer review is being conducted. In general, each of the forms of peer review require that some sort of review package is assembled and distributed. This package commonly contains a summary of the requirement(s) which are the basis for the product being reviewed. Other common inputs are differentiated by the stage of the lifecycle currently in process. For example, input to a peer review during the coding phase would consist of program listings, design specifications, programming standards and a summary of results from the design peer review previously held on the same product. Common input to particular forms of peer review are described below. (A summary of the methodology for each of these reviews appears in Section 5.)

a. Code-Reading Review.

- o Component requirements
- o Design specifications
- o Program listings
- o Programming standards

b. Round-Robin Reviews.

- o Component requirements
- o Design or code specifications
- o Program listings (if during coding phase)

c. Walkthrough.

- o Component requirements
- o Design or code specifications
- o Program listings (if coding phase walkthrough)
- o Product standards
- o Back-up documentation (i.e., flowcharts, HIPO charts, data dictionaries, etc.)
- o Question list (derived by participants prior to review)

d. Inspections.

- o Component requirements
- o Design or code specifications

- o Program listings (if during coding phase)
- o Product standards
- o Back-up documentation
- o Checklist (containing descriptions of particular features to be evaluated)

4.20.4. Information output. The output from a peer review varies by form of review. One output common to each form of a peer review is a decision or consensus about the product under review. This is usually in the form of a group approval of the product as is, an approval with recommended modifications, or a rejection (and rescheduled review date).

Specific output from peer reviews described in Section 5 are as follows:

a. Code Reading Review and Round-Robin Review.

- o Informal documentation of detected problems
- o Recommendation to accept or reject reviewed product
- o Discrepancy List

b. Walkthrough.

- o Action List (formal documentation of problems)
- o Walkthrough Form (containing review summary and group decision)

c. Inspection.

- o Inspection Schedule and Memo (defining individual roles, responsibilities, agenda and schedule)
- o Problem Definition Set
- o Summary report (documenting error correction status and related statistics on the errors)
- o Management report (describing errors, problems and component status)

4.20.5. Outline of method. The peer review methodology and participant responsibilities vary by form of review. Summaries of these methodologies are provided in the later part of this section. However, there are a few features common to each methodology.

For example, most peer reviews are not attended by management. (An exception is made in circumstances where the project manager is also a designer, coder or tester -- usually on very small projects.) The presence of management tends to inhibit participants, since they feel that they are personally being evaluated. This would be contrary to the intent of peer reviews -- that of studying the product itself.

Another common feature is the assembly and distribution of project review materials prior to the conduct of the peer review. This allows participants to spend some amount of time reviewing the data to become better prepared for the review.

At the end of most peer reviews the group arrives at a decision about the status of the review product. This decision is usually communicated to management.

Most reviews are conducted in a group organization as opposed to individually by participants or by the project team itself. While this may seem an obvious feature, it bears some discussion. Most organizations doing software development and/or maintenance employ some variation of a team approach. Some team organizations are described below.

- o Conventional Team - A senior programmer directs the efforts of one or more less experienced programmers.
- o Egoless Team - Programmers who are of about equal experience share product responsibilities.
- o Chief Programmer Team - A highly qualified senior programmer leads the efforts of other team members for which specific roles and responsibilities have been assigned (i.e., back-up programmer, secretary, librarian, etc.).

The group which participates in the peer review is not necessarily the same as the team organized to manage and complete the software product. The review group is likely to be composed of a subset of the project team plus other individuals as required by the form of review being held and the stage of the lifecycle in process. The benefits of peer reviews are unlikely to be attained if the group acts separately, without some designated responsibilities. Some roles commonly used in review groups are described below. These roles are not <u>all</u> employed in any one review but represent a list.

- o Group/Review leader - the individual designated by management with planning, detecting, organizing and coordinating responsibilities. Usually has responsibilities after the review to ensure that recommendations are implemented.
- o Designer - the individual responsible for the specificatior of the product and a plan for its implementation.
- o Implementor - the individual responsible for developing the product according to the plan detailed by the designer.
- o Tester - the individual responsible for testing the product as developed by the implementer.
- o Coordinator - the individual designated with planning, directing, organizing and coordinating responsibilities.
- o Producer - the individual whose product is under review.
- o Recorder - the individual responsible for documenting the review activities during the review.
- o User Representative - the individual responsible for ensuring that the user's requirements are addressed.
- o Standards Representative - the individual responsible for ensuring that product standards are conformed to.
- o Maintenance Representative - the individual who will be responsible for updates or corrections to the installed product.
- o Others - individuals with specialized skills or responsibilities which contribute during the peer review.

While the forms of peer reviews have some similarities and generally involve designation of participant roles and responsibilities, they are different in application. The remainder of this section will summarize the application methods associated with the forms of peer reviews previously introduced.

a. Code Reading Review. Code reading is line-by-line study and evaluation of program source code. It is generally performed on source code which has been compiled and is free of syntax errors. However, some organizations practice code reading on uncompiled source listings or hand written code on coding sheets in order to remove syntax and logic errors prior to code entry. Code reading is commonly practiced on top-down, structured code and becomes cost ineffective when performed on unstructured code.

The optimum size of the code reading review team is three to four. The producer sets up the review and is responsible for team leadership. Two or three programmer/analysts are selected by the producer based upon their experience, responsibilities with interfacing programs, or other specialized skill.

The producer distributes the review input (see section 4.20.3) about two days in advance. During the review the producer and the reviewers go through each line of code checking for features which will make the program more readable, usable, reliable and maintainable. Two types of code reading may be performed: reading for understanding and reading for verification. Reading for understanding is performed when the reader desires an overall appreciation of how the program module works, its structure, what functions it performs, and whether it follows established standards. Assuming that figure 4.20.5-1 depicts the structure of a program component, a reviewer reading for understanding would review the modules in the the following order: 1.0, 2.0, 2.1, 2.2, 3.0, 3.1, 3.2, 3.3.

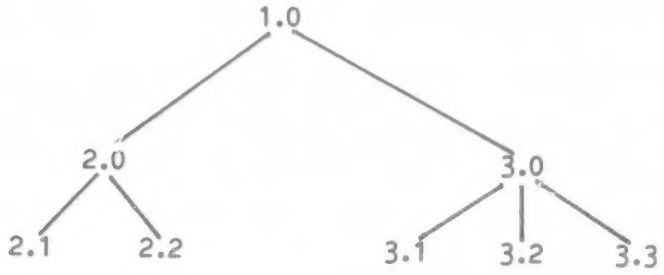

Figure 4.20.5-1 A Program Structure

In contrast to this top-to-bottom approach, reading for verification implies a bottom-up review of the code. The component depicted above would be perused in the following order: 3.3, 3.2, 3.1, 3.0, 2.2, 2.1, 2.0, 1.0. In this manner it is possible to produce a dependency list detailing parameters, control switches, table pointers, and internal and external variables used by the component. The list can then be used to ensure hierarchical consistency,

data availability, variable initiation, etc. Reviewers point out any problems or errors detected while reading for understanding or verification during the review.

The team then makes an informal decision about the acceptability of the code product and may recommend changes. The producer notes suggested modifications and is responsible for all changes to the source code. Suggested changes are evaluated by the producer and need not be implemented if the producer determines that they are invalid.

There is no mechanism to ensure that change is implemented or to follow up on the review.

b. Round-Robin Review. A round-robin review is a peer review where each participant is given an equal and similar share of the product being reviewed to study, present, and lead in its evaluation.

A round-robin review can be given during any phase of the product lifecycle and is also useful for documentation review. In addition, there are variations of the round-robin review which incorporate some of the best features from other peer review forms but continue to use the alternating review leader approach. For example, during a round-robin inspection, each item on the inspection checklist is made the responsibility of alternating participants.

The common number of people involved in this type of peer review is four to six. The meeting is scheduled by the producer, who also distributes some high level documentation as described in section 3. The producer will either be the first review leader or will assign this responsibility to another participant. The temporary leader will guide the other participants (who may be implementers, designers, testers, users, maintenance representatives, etc.) through the first unit of work. This unit may be a module, paragraph, line of code, inspection item, or other unit of manageable size. All participants (including the leader) have the opportunity to comment on the unit before the next leader begins the evaluation of the next unit. The leaders are responsible for noting major comments raised about their piece of work. At the end of the review all the major comments are summarized and the group decides whether or not to approve the product. No formal mechanism for review follow up is used.

c. Walkthroughs. This type of peer review is more formal than the code reading review or round-robin review. Distinct roles and responsibilities are assigned prior to review. Prereview preparation is greater, and a more formal approach to problem documentation is stressed. Another key feature of this review is that it is presented by the producer. The most common walkthroughs are those held during design and code, yet recently they are being applied to specifications documentation and test results.

The producer schedules the review and assembles and distributes input as described in section 3. In most cases the producer selects the walkthrough participants (although sometimes this is done by management) and notifies them

of their roles and responsibilities. The walkthrough is usually conducted with less than seven participants and lasts not more than 2 hours. If more time is needed a break must be given or the product should be reduced in size. Roles usually included in a walkthrough are producer, coordinator, recorder, and representatives of user, maintenance and standards organizations.

The review is opened by the coordinator, yet the producer is responsible for leading the group through the product. In the case of design and code walkthrough, the producer simulates the operation of the component, allowing each participant to comment based upon his area of specialization. A list of problems is kept and at the end of the review each participant signs the list or other walkthrough form indicating whether the product is accepted as-is, accepted with recommended changes, or rejected. Suggested changes are made at the discretion of the producer. There is no formal means of follow up on the review comments. However, if the walkthrough review is used for products as they evolve during the lifecycle (i.e., specification, design, code and test walkthrough), comments from past reviews can be discussed at the start of the next review.

d. Inspections. Inspections are the most formal, commonly used form of peer review. The key feature of an inspection is that it is driven by the use of checklists to facilitate error detection. These checklists are updated as statistics indicate that certain types of error are occurring more or less frequently than in the past. The most commonly held types of inspections are conducted on the product design and code, although inspections may be used during any lifecycle phase. Inspections should be short since they are often quite intensive. This means that the product component to be reviewed must be of small size. Specifications or design which will result in 50-100 lines of code are normally manageable. This translates into an inspection of 15 minutes to 1 hour, although complex components may require as much as 2 hours. In any event, inspections of more than 2 hours are generally less effective and should be avoided. Two or three days prior to the inspection the producer assembles input as described in section 3 and gives it to the coordinator for distribution. Participants are expected to study and make comments on the materials prior to the review.

The review is lead by a participant other than the producer. Generally, the individual who will have the greatest involvement in the next phase of the product lifecycle is designated as reader. For example, a requirements inspection would likely be lead by a designer, a design review by an implementer, and so forth. The exception to this occurs for a code inspection which is lead by the designer. The inspection is organized and coordinated by an individual designated as the group leader or coordinator.

The reader goes through the product component, using the checklist as a means to identify common types of errors as well as standards violations. A primary goal of an inspection is to identify items which can be modified to make the component more understandable, maintainable, or usable. Participants (identified earlier in this section) discuss any issues which they identified in preinspection study.

At the end of the inspection an accept/reject decision is made by the group and the coordinator summarizes all the errors and problems detected and provides this list to all participants. The individual whose work was under review (designer, implementer, tester, etc.) uses the list to make revisions to the component. When revisions are implemented, the coordinator and producer go through a minireview using the problem list as a checklist.

The coordinator then completes Management and Summary Reports. The Summary report is used to update checklists for subsequent inspections.

4.20.6. Example. The following is an example describing a code reading review.

Three days prior to estimated completion of coding, the producer of a program component begins preparation for a code reading review. The component is composed of 90 lines of FORTRAN code and associated comments. The producer obtains copies of the source listing, and requirements and design specifications for the component and distributes them to three peers, notifying them of the review date and place.

Each reviewer reads the code for general understanding, reviewing a major function and its supporting functions prior to reviewing the next major function (see section 5).

One reviewer notes an exception to the programming standards. Another thinks that the data names are not meaningful. The third has found several comments which inaccurately represent the function they describe. Each reviewer makes a note of these points as well as any comments about the structure of the component. Next, the requirements are studied to ensure that each requirement is addressed by the component. It appears that the requirements have all been met.

The code reading review is led by the producer. After a brief description of the component and its interfaces, the producer leads the reviewers through the code. Rather than progressing through the component from top to bottom, the decision is made to perform code-reading from the bottom up. This form of code-reading is used to verify the component's correctness (see section 5).

As the code is being perused, one of the reviewers is made responsible for keeping a dependency list. As each variable is defined, referenced, or modified, a notation is made on the list.

The verification code reading uncovers the use of a variable prior to its definition. This error is documented on an error list by the producer. In addition, each of the problems detected earlier during the code reading (as performed by each individual) is discussed and documented.

At the end of the review, the error list is summarized to the group by the producer. Since none of the problems are major, the participants agree to accept the code with the agreed to minor modifications. The producer then uses the error/problem list for reference when making modifications to the component.

4.20.7. Effectiveness. Studies have been conducted which identify the following qualitative benefits the forms of peer reviews.

- o higher status visibility,
- o decreased debugging time,
- o early detection of design and analysis errors which would be much more costly to correct in later development phases,
- o identification of design or code inefficiencies,
- o ensuring adherence to standards,
- o increased program readability,
- o increased user satisfaction,
- o communication of new ideas or technology,
- o increased maintainability.

Little data is available which identifies the quantitative benefits attributable to the use of a particular form of peer review. However, one source estimates that the number of errors in production programs was reduced by a factor of ten by utilizing walkthroughs. Another source estimates that a project employing inspections achieved 23% higher programmer productivity than with walkthroughs. No data was available indicating the amount of increased programmer productivity attributable to the inspections alone.

4.20.8. Applicability. Peer reviews are applicable to large or small projects during all development phases and are not limited by project type or complexity.

4.20.9. Learning. None of the peer reviews discussed require extensive training to implement. They do require familiarity with the concept and methodology involved. Experience has shown that peer reviews are most successful when the individual with responsibility for directing the review is knowledgeable about the process and its intended results.

4.20.10. Costs. The reviews require no special tools or equipment. The main cost involved is that of human resources. If the reviews are conducted in accordance with the resource guidelines expressed in most references, the cost depends upon the number of reviews required. Most references suggest that peer reviews should be no longer than 2 hours, preferrably 1/2 to 1 hour. Preparation time can amount to as little as 1/2 hour and should not require longer than 1/2 day per review.

4.20.11. References.

(1) "Code Reading Structured Walk-Throughs and Inspections", IBM IPTO Support Group,World Trade System Center, Postbus 60, Zoetermeer, Netherlands, March 1976.

(2) FAGEN, M.E., "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal,No.3,1976.

(3) YOURDON, E., "Structured Walkthroughs",Yourdon Inc., 1977.

(4) FREEDMAN, D.P. and WEINBERG, G.M., "Ethno - Technical Review Handbook," 1977, Ethnotech, Inc.

(5) DALY, E.B., "Management of Software Development", IEEE Transactions on Software Engineering, May 1977.

(6) SHNEIDERMAN, Ben, "Software Psychology - Human Factors in Computer and Information Systems," Winthrop Publishing, 1980.

4.21.1. Name. Physical Units Checking.

4.21.2. Basic features. Many (scientific, engineering, and control) programs perform computations whose results are interpreted in terms of physical units, such as feet, meters, watts, and joules. Physical units checking enables specification and checking of units in program computations, in a manner similar to dimensional analysis. Operations between variables which are not commensurate, such as adding gallons and feet, are detected.

4.21.3. Information input. Units checking requires three things to be specified within a program: 1) the set of elementary units used (such as feet, inches, acres), 2) relationships between the elementary units (such as feet = 12 inches, acre = 43,560 square feet), and 3) the association of units with program variables. The programming language used must support such specifications, or the program must be preprocessed by a units checker.

4.21.4. Information output. The information output depends upon the specific capabilities of the language processor or preprocessor. At a minimum, all operations involving variables which are not commensurate are detected and reported. If variables are commensurate, but not identical (i.e., they are the same type of quantity, such as units of length, but one requires application of a scaler multiplier to place it in the same units as the other), the system may insert the required multiplication into the code, or may only report what factor must be applied by the programmer.

4.21.5. Outline of method. The specification of the input items is the extent of the actions required by the user. Some systems may allow the association of a units expression with an expression within the actual program. Thus, one may write LOTSIZE (LENGTH * WIDTH * square feet) as a boolean expression, where the product of LENGTH and WIDTH must be in units of square feet. The process of ensuring that LENGTH * WIDTH is in square feet is the responsibility of the processing system.

4.21.6. Example. A short program in Pascal-like notation is shown for computing the volume and total surface area of a right circular cylinder. The program requires as input the radius of the circular base and the height of the cylinder. Because of peculiarities in the usage environment of the program, the radius is specified in inches, the height in feet; volume is required in cubic feet, and the surface area in acres. Several errors are present in the program, all of which would be detected by the units checker.

In the following, comments are made explaining the program, the errors it contains, and how they would be detected. The comments are keyed by line number to the program.

Line Number	Comment
2	All variables in the program which are quantities will be expressed in terms of these basic units.
3	These are the relationships between the units known to the units checker.
5-10	Variable radius is in units of inches, height is in units of feet, and so forth.

12 Input values are read into variables radius and height.
13 Lateral surface must be expressed in square feet. (RADIUS/12) is in feet, and can be so verified by the checker.
15 Lateral-surface and top-surface are both expressed in square feet, thus their sum is in square feet, also. Area is expressed in acres, however, and the checker will issue a message to the effect that though the two sides are commensurate the conversion factor of 43,560 was omitted from the right side of the assignment.
16 The checker will detect that the two sides of the assignment are not commensurate. The right side is in units of feet quadrupled, the left is in feet cubed.

```
(1) program cylinder (input, output);
(2) elementary units inches, feet, acre;
(3) units relationships feet = 12 inches; acre = 43,560 feet**2;
(4) constant pi = 3.1415927
(5) var radius (inches),
(6)     height (feet),
(7)     volume (feet**3),
(8)     area (acre),
(9)     lateral-surface (feet**2),
(10)    top-surface (feet**2):  real;
(11) begin
(12)    read (radius, height);
(13)    lateral-surface := 2*PI*(radius/12)*height;
(14)    top-surface := PI* (radius/12)**2
(15)    area := lateral-surface + 2* top-surface;
(16)    volume := PI *((radius**3)*height);
(17)    write (area, volume);
(18) end;
```

4.21.7. Effectiveness. The effectiveness of units checking is limited only by the capabilities of the units processor.

Simple units checkers may only be able to verify that two variables are comunsurate, but not determine if proper conversion factors have been applied. That is, a relationship such as 12 inches = feet may not be fully used in checking the computations in a statement, such as line 13 of the example. There we asserted that (radius/12) would be interpreted as converting inches to feet. The checker may not support this kind of analysis, however, to avoid ambiguities with expressions such as "one-twelfth of the radius."

4.21.8. Applicability. Certain application areas, such as engineering and scientific, often deal with physical units. In others, however, it may be difficult to find analogies to physical units. In particular, if a program deals only in one type of quantity, such as dollars, the technique would not be useful.

Units checking can be performed during all stages of software development, beginning with requirements specifications.

4.21.9. Learning. Dimensional analysis is commonly taught in first year college physics on statics; conversion from English to metric units is common throughout society. Direct application of these principles in programming, using a units checker, should require no additional training beyond understanding the capabilities of the specific units checker and the means for specifying units-related information.

4.21.10. Cost. If the units checking capabilities are incorporated directly in a compiler its usage cost should be negligible. If a preprocessor is used, such systems are typically much slower than a compiler (perhaps operating at 1/10 compilation speed), but only a single analysis of the program is required. The analysis is only repeated when the program is changed.

4.21.11. References.

(1) KARR, Michael and LOVEMAN III, David B., "Incorporation of Units into Programming Languages", CACM, Vol. 21, No.5, pp. 385-391, May 1978.

4.22.1. Name: Regression Testing

4.22.2. Basic features. Regression testing is a technique whereby spurious errors caused by system modifications or corrections may be detected.

4.22.3. Information input. Regression testing requires that a set of system test cases be maintained and available throughout the entire life of the system. The test cases should be complete enough so that all of the system's functional capabilities are thoroughly tested. If available, acceptance tests should be used to form the base set of tests.

In addition to the individual test cases themselves, detailed descriptions or samples of the actual expected output produced by each test case must also be supplied and maintained.

4.22.4. Information output. The output from regression testing is simply the output produced by the system from the execution of each of the individual test cases. When the output from previous acceptance tests has been kept, additional output from regression testing should be a comparison of the before and after executions.

4.22.5. Outline of method. Regression testing is the process of retesting the system in order to detect errors which may have been caused by program changes. The technique requires the utilization of a set of test cases which have been developed (ideally, using functional testing) to test all of the system's functional capabilities. If an absolute determination of portions of the system which can potentially be affected by a given change can be made, then only those portions need to be tested. Associated with each test case is a description or sample of the correct output for that test case. When the tests have been executed, the actual output is compared with the expected output for correctness. As errors are detected during the actual operation of the system which were not detected by regression testing, a test case which could have uncovered the error should be constructed and included with the existing test cases.

Although not required, tools can be used to aid in performing regression testing. Automatic test harnesses can be used to assist the managing of test cases and in controlling the test execution. File comparators can often be useful in verifying actual output with expected output. Assertion processors are also useful in verifying the correctness of the output for a given test.

4.22.6. Example.

a. Application. A transaction processing system contains a dynamic data field editor which provides a variety of input/output field editing capabilities. Each transaction is comprised of data fields as specified by a data element dictionary entry. The input and output edit routine used by each data field is specified by a fixed identifier contained in a data field descriptor in the dictionary entry. When a transaction is input, each field is edited by the appropriate input editor routine as specified in the dictionary entry. Output editing consists of utilizing output editor routines to format the output.

b. Error. An input edit routine to edit numeric data fields was modified to perform a fairly restrictive range check needed by a particular transaction program. Current system documentation indicated that this particular edit routine was only being used by that single transaction program. However, the documentation was not up-to-date in that another, highly critical, transaction program also used the routine, often with data falling outside of the range check needed by the other program.

c. Error discovery. Regression testing would uncover the error given that a sufficient set of functional tests were used for performing the testing. If only the transaction program for which the modification was made were tested, the error would not have been discovered until actual operation.

4.22.7. Effectiveness. The effectiveness of the technique depends upon the quality of the data used for performing the regression testing. If functional testing, i.e. tests based on the functional requirements, is used to create the test data, the effectiveness is highly effective. The burden and expense associated with the technique, particularly for small changes, can appear to be prohibitive. It is, however, often quite straightforward to determine which functions can be potentially affected by a given change. In such cases, the extent of the testing can be reduced to a more tractable size.

4.22.8. Applicability. This method is generally applicable.

4.22.9. Learning. No special training is required in order to apply the technique. If tools are used in support of regression testing, however, knowledge of their use will be required. Moreover, successful application of the technique will require establishment of procedures and the management control necessary to ensure adherence to those procedures.

4.22.10. Costs. Since testing is required as a result of system modifications anyway, no additional burden need result because of the method (assuming that only the necessary functional capabilities are retested). The use of tools, however, to support it could increase the cost but it would also increase its effectiveness.

4.22.11. References.

(1) PANZL, David J., "Automatic Software Test Drivers," Computer, April 1978.

(2) FISHER,K.F.,"A Test Case Selection Method for the Validation of Software Maintenance Modification",IEEE COMPSAC,1977.

(3) FISHER,K.F.,RAJI,F.,and CHRUSCICK,A.,"A Methodology for Re-testing Modified Software",National Telecommunications Conference,New Orleans,LA.,Nov. 1981.

4.23.1. Name. Requirements Analyzer.

4.23.2. Basic features. The requirements for a system will normally be specified using some formal language which may be graphical and/or textual in nature. A requirements analyzer can check for syntactical errors in the requirements specifications and then produce a useful analysis of the relationships between system inputs, outputs, processes, and data. Logical inconsistencies or ambiguities in the specifications can also be identified by the requirements analyzer.

4.23.3. Information input. The form and content of the input will vary greatly for different requirements languages. Generally, there will be requirements regarding what the system must produce (outputs) and what types of inputs it must accept. There will usually be specifications describing the types of processes or functions which the system must apply to the inputs in order to produce the outputs. Additional requirements may concern timing and volume of inputs, outputs, and processes as well as performance measures regarding such things as response time and reliability of operations. The form of the inputs to the requirements analyzer is specified by the requirements specification language and varies considerably for different languages. In some cases all inputs are textual, whereas some languages utilize all graphical inputs from a display terminal (e.g., boxes might represent processes and arrows between boxes might represent information flow).

4.23.4. Information output. Nearly all analyzers produce error reports showing syntactical errors or inconsistencies in the specifications. For example, the syntax may require that the outputs from a process at one level of system decomposition must include all outputs from a decomposition of that process at a more detailed level. Similarly, for each system output there should be a process which produces that output. Any deviations from these rules would result in error diagnostics.

Each requirements analyzer produces a representation of the system which indicates static relationships among system inputs, outputs, processes, and data. Some analyzers also represent dynamic relationships and provide an analysis of them. This may be a precedence relationship, e.g., process A must execute before process B. It may also include information regarding how often a given process must execute in order to produce the volume of output required. Some analyzers produce a detailed representation of relationships between different data items. This output can sometimes be used for developing a data base for the system. A few requirements analyzers go even further and provide a mechanism for simulating the requirements using the generated system representation including the performance and timing requirements.

4.23.5. Outline of method. The user must provide the requirements specifications as input for the analyzer. The analyzer carries out the analysis in an automated manner and provides it to the user who must then interpret the results. Often the user can request selected types of outputs, e.g., an alphabetical list of all the processes or a list of all the data items of a given type. Some analyzers can be used either interactively or in

a batch mode. Once the requirements specifications are considered acceptable, a few analyzers provide the capability for simulating the requirements. It is necessary that the data structure and data values generated from the requirements specifications be used as input to the simulation, otherwise the simulation may not truly represent the requirements.

4.23.6. Example. Suppose that a process called PROCESS B produces two files named H2 and H3 from an input file name M2. (The purposes of the files are irrelevant to the discussion.) Suppose also that PROCESS D accepts Files H2 and H3 as input and produces Files J3 and J6 output. In addition, PROCESS G is a subprocess of PROCESS D and it accepts File H3 as input and produces File J6. Then the pseudo specification statements, figure 4.23.6-1, might be used to describe the requirements. (Note that these requirements are close to design, but this is often the case.)

```
PROCESS B
  .
  .
  .
USES FILE M2
PRODUCES FILES H2, H3

PROCESS D
  .
  .
  .
USES FILES H2, H3
PRODUCES FILES J3, J6

PROCESS G
  .
  .
  .
SUBPROCESS OF PROCESS D
USES FILE H3
PRODUCES FILE J6
```

Figure 4.23.6-1 Requirements Specification Statements

The requirements specifications imply a certain precedence of operations, e.g., PROCESS D cannot execute until PROCESS B has produced files H2 and H3. Detailed descriptions of what each process does would normally be included, but are omitted for brevity. The requirements analyzer would probably generate a diagnostic since the statement for PROCESS D fails to indicate that it includes the subprocess G. A diagnostic would also be generated unless there are other statements which specify that file M2, needed by PROCESS B, is available as an existing file or else is produced by some other process. Similarly, other processes must be specified which use files J3 and J6 as input unless they are specified as files to be output from the system. Otherwise, additional diagnostics would be generated. It can be seen that some of the checks are similar to data flow analysis for a computer program.

However, for large systems the analysis of requirements becomes very complex if requirements for timing and performance are included, and if timing and volume analysis are to be carried out. (Volume analysis is concerned with such things as how often various processes must execute if the system is to accept and/or produce a specified volume of data in a single given period of time.)

4.23.7. Effectiveness. Some requirements analyzers are very effective for maintaining accurate requirements specifications. For large systems with a large number of requirements they are essential. On the other hand, most existing requirements analyzers are rather expensive to obtain and use, and they may not be cost effective for development of small systems.

4.23.8. Applicability. Requirements analyzers are applicable for use in developing most systems. They are particularly useful for analysis of requirements for large and complex systems.

4.23.9. Learning. Most requirements analyzers require a considerable amount of training of personnel.

4.23.10. Cost. Most requirements analyzers are expensive to obtain and use. They generally require a large amount of storage within a computer and so can only be used on large computers.

4.23.11. References.

(1) ALFORD, Mack W., "A Requirements Engineering Methodology for Real-Time Processing Requirements," TRW Software Series, TRW-SS-76-07, Systems Engineering and Integration Division, September 1976.

(2) TEICHROEW, Daniel, "A Survey of Languages for Stating Requirements for Computer-Based Information Systems," The University of Michigan, Proceedings of the Fall Joint Computer Conference, 1972, pp. 1203-1224.

4.24.1. Name. Requirements Tracing.

4.24.2. Basic features. Requirements tracing provides a means of verifying that the software of a system addresses each requirement of that system and that the testing of the software produces adequate and appropriate responses to those requirements.

4.24.3. Information input. The information needed to perform requirements tracing consists of a set of system requirements and the software which embodies the capability to satisfy the requirements.

4.24.4. Information output. The information output by requirements tracers is the correspondence found between the requirements of a system and the software that is intended to realize these requirements.

4.24.5. Outline of method. Requirements tracing generally serves two major purposes. The first is to ensure that each specified requirement of a system is addressed by an identifiable element of the system software. The second is to ensure that the testing of that software produces results which are adequate responses in satisfying each of these requirements.

A common technique used to assist in making these assurances is the use of test evaluation matrices. These matrices represent a visual scheme of identifying which requirements of a system have been adequately and appropriately addressed and which have not. There are two basic forms of test evaluation matrices. The first form identifies a mapping that exists between the requirement specifications of a system and the modules of that system. This matrix determines whether each requirement is realized by some module in the system, and, conversely, whether each module is directly associated with a specific system requirement. If the matrix reveals that a requirement is not addressed by any module, then that requirement has probably been overlooked in the software design activity. If a module does not correspond to any requirement of the system, then that module is superfluous to the system. In either case, the design of the software must be further scrutinized, and the system must be modified accordingly to effect an acceptable requirements-design mapping.

The second form of a test evaluation matrix provides a similar mapping, except the mapping exists between the modules of a system and the set of test cases performed on the system. This matrix determines which modules are invoked by each test case. Used with the previous matrix, it also determines which requirements will be demonstrated to be satisfied by the execution of a particular test case in the test plan. During actual code development, it can be used to determine which requirement specifications will relate to a particular module. In this way, it is possible to have each module print out a message during execution of a test indicating which requirement is referenced by the execution of this module. The code module itself may also contain comments about the applicable requirements.

If these matrices are to be used most effectively in a requirements tracing activity, the two matrices should be used together. The second matrix is built prior to software development. After the software has been developed

and the test cases have been designed (based upon this matrix), it is necessary to determine whether the execution of the test plan will actually demonstrate satisfaction of the requirements of the software system. By analyzing the results of each test case, the first matrix can be constructed to determine the relationship that exists between the requirements and software reality.

The first matrix is mainly useful for analyzing the functional requirements of a system. However, the second matrix is also useful in analyzing the performance, interface, and design requirements of the system, in addition to the functional requirements. Both are often used in support of a more general requirements tracing activity, that of preliminary and critical design reviews. This is a procedure used to ensure verification of the traceability of all the above mentioned requirements to the design of the system. In addition to the use of test evaluation matrices, these design reviews may include the tracing of individual subdivisions in the software design document back to applicable specifications made in the requirements document. This is a constructive technique used to ensure verification of requirements traceability.

4.24.6. Example.

a. Application. A new payroll system is to be tested. Among the requirements of this system is the specification that all employees of age 65 or older:

1. receive semi-retirement benefits, and

2. have their social security tax rate readjusted.

To ensure that these particular requirements are appropriately addressed in the system software, test evaluation matrices have been constructed and filled out for the system.

b. Error. An omission in the software causes the social security tax rate of individuals of age 65 or older to remain unchanged.

c. Error discovery. The test evaluation matrices reveal that the requirement that employees of age 65 or older have their social security tax rate adjusted has not been addressed by the payroll program. No module in the system had been designed to respond to this specification. The software is revised accordingly to accommodate this requirement, and a test evaluation matrix is used to ensure that the added module is tested in the set of test cases for the system.

4.24.7. Effectiveness. Requirements tracing is a highly effective technique in discovering errors during the design and coding phases of software development. This technique has proven to be a valuable aid in verifying the completeness, consistency, and testability of software. If a system requirement is modified, it also provides much assistance in retesting software by clearly indicating which modules must be rewritten and retested. Requirements tracing can be a very effective technique in detecting errors

early in the software development cycle which could otherwise prove to be very expensive if discovered later.

4.24.8. Applicability. This technique is generally applicable in large or small system testing and for all types of computing applications. However, if the system requirements themselves are not clearly specified and documented, proper requirements tracing can be very difficult to accomplish in any application.

4.24.9. Learning. Knowledge and a clear understanding of the requirements of the system is essential. More complex systems will result in a corresponding increase in required learning.

4.24.10. Costs. No special tools or equipment are needed to carry out this technique if done manually. The major cost in requirements tracing is that associated with human labor expended. Requirements tracing is often a feature of requirements analyzers which are expensive to obtain and use.

4.24.11. References.

(1) "THREADS: A Functional Approach to Project Control,"Computer Sciences Corp., El Segundo, California, 1975.

(2) HETZEL, W.C., "An Experimental Analysis of Program Verification Methods," Ph.D. Thesis, University of North Carolina, 1976.

4.25.1. Name. Software monitors.

4.25.2. Basic features. These tools monitor the execution of a program in order to locate and identify possible areas of inefficiency in the program. Execution data is obtained while the program executes in its normal environment. At the end of execution, reports are generated by the monitor summarizing the resource usage of the program.

4.25.3. Information input. Software monitors require as input the program source code to be executed and any data necessary for the program to run. Certain commands must also be provided by the user in specifying the information to be extracted by the monitor and in specifying the format of the generated output reports. These commands may specify:

- o what is to be measured (e.g., execution times, I/O usage, core usage, paging activity, program waits),
- o the specific modules to be monitored,
- o the frequency that data is to be extracted during program execution (sampling interval),
- o the titles, headings, content of each output report,
- o the units used to construct graphs,
- o whether the graphs are to be displayed as plots or histograms.

4.25.4. Information output. The output of a software monitor is a set of one or more reports describing the execution characteristics of the program. Information that may be contained in these reports is given below.

- o A summary of all the sample counts made during data extraction, e.g., the number of samples taken where the program was executing instructions, waiting for the completion of an I/O event, or otherwise blocked from execution.
- o A summary of the activity of each load module.
- o An instruction location graph that gives the percentage of time spent for each group of instructions partitioned in memory.
- o A program timeline that traces the path of control through time.
- o A control passing summary that gives the number of times control is passed from one module to another.
- o A wait profile showing the number of waits encountered for each group of instructions.
- o A paging activity profile that displays pages-in and pages-out for each group of instructions.

This information is often represented in histograms and/or plotted graphs.

4.25.5. Outline of method. Software monitors typically consist of two processing units. The first unit runs the program being monitored and collects data concerning the execution characteristics of the program. The second unit reads the collected data and generates reports from it.

A software monitor monitors a program by determining its status at periodic intervals. The period between samples is usually controlled through an elapsed interval timing facility of the operating system. Samples are taken

from the entire address range addressable by the executing task. Each sample may contain an indication of the status of the program, the load module in which the activity was detected, and the absolute location of the instruction being executed. Small sample intervals increase sampling accuracy but result in a corresponding increase in the overhead required by the CPU.

The statistics gathered by the data extraction unit are collected and summarized in reports generated by the data analysis unit. References to program locations in these reports will be in terms of absolute addresses. However, in order to relate the absolute locations to source statements in the program, the reports also provide a means to locate in a compiler listing the source statement that corresponds to that instruction. In this way, sources of waits and program locations that use significant amounts of CPU time can be identified directly in the source code; any performance improvements to the program will occur at these identified statements.

Software monitors are similar to another tool used to monitor program execution, test coverage analyzers. Test coverage analyzers keep track of and report on the number of times that certain elementary program constructs in a program have been traversed during a sequence of tests. During the monitoring of a program, both tools count the frequency that certain events occur. After program execution, both generate reports summarizing the data collected. However, because these tools serve different functions, they are different in their techniques of gathering information and in the type of information each collects. Test coverage analyzers are used to measure the completeness of a set of program tests, while software monitors measure the resource usage of a program as a means of evaluating program efficiency. As an evaluation of program efficiency requires consideration of execution time expenditure, software monitors utilize a strict timing mechanism during the collection of data. This is absent in monitors such as test coverage analyzers which are not used to evaluate program performance.

4.25.6. Example.

a. Application. A program that solves a set of simultaneous equations is constructed. The program first generates a set of coefficients and a right hand side for the system being solved. It then proceeds to solve the system and output the solution.

b. Error. In the set of calculations required to solve the system, a row of coefficients is divided by a constant and then subtracted from another row of coefficients. The divisions are performed within a nested DO-loop but should be moved outside the innermost loop, as the dividend and divisors within the loop do not change.

c. Error discovery. The performance of the program is evaluated through the use of a software monitor. Examination of the output reveals that the program spends almost 85% of its time in a particular address range. Further analysis shows that 16.65% of all CPU time is used by a single instruction. A compiler listing of the program is used to locate the source statement that generated this instruction, which is found to be the statement containing the division instruction. Once the location of the inefficiency is

discovered, it is left to the programmer to determine whether and how the code can be optimized.

4.25.7. Effectiveness. Software monitors are valuable tools in identifying performance problems in a program. Their overall effectiveness, however, is dependent upon the quality of their use.

4.25.8. Applicability. Software monitors can be applied to any kind of program in any programming language.

4.25.9. Learning. There are no special learning requirements for the use of software monitors. In order to use the tools effectively, however, the input parameters to the monitor must be carefully selected in determining the most relevant reports to be generated. Once the areas of a program which are most inefficient have been identified, it requires skill to modify the program to improve its performance.

4.25.10. Costs. The largest cost in using a software monitor is that incurred by the CPU to extract the data during execution. In one implementation, extraction of data resulted in an increase of user program CPU time by 1% to 50%. Storage requirements also increase in order to provide memory for diagnostic tables and the necessary program modules of the tool.

4.25.11. References.

(1) "Problem Program Evaluator (PPE) User Guide," Boole and Babbage, Inc., Sunnyvale, California, March, 1978.

(2) RAMAMOORTHY, C.V. and KIM, K.H., "Software Monitors Aiding Systematic Testing and Their Optional Placement," Proceedings of the First National Conference on Software Engineering, IEEE Catalog No. 75CH0992-8C, September, 1975.

4.26.1. Name. Specification-Based Functional Testing.

4.26.2. Basic features. Functional testing can be used to generate system test data from the information in requirements and design specifications. It is used to test both the overall functional capabilities of a system and functions which originate during system design.

4.26.3. Information input.

a. Data information. The technique requires the availability of detailed requirements and design specifications and, in particular, detailed descriptions of input data, files and data bases. Both the concrete and algebraic abstract properties of all data must be described. Concrete properties include type, value ranges and bounds, record structures, and bounds on file data structure and data base dimensions. Abstract properties include subclasses of data that correspond to different functional capabilities in the system and subcomponents of compound data items that correspond to separate subfunctional activities in the system.

b. Function information. The requirements and design specifications must also describe the different functions implemented in the system.

Requirements functions correspond to the overall functional capabilities of a system or to subfunctions which are visible at the requirements stage and are necessary to implement overall capabilities. Different overall functional capabilities correspond to conceptually distinct classes of operations that can be carried out using the system. Different kinds of subfunctions can also be identified. Process descriptions in structured specifications, for example, describe data transformations which are visible at requirements time and which correspond to requirements subfunctions. Requirements subfunctions also occur implicitly in data base schemata. Data base functions are used to reference, update and create data bases and files.

The designer of a system will have to invent both general and detailed functional constructs in order to implement the functions in requirements specifications. Structured design techniques are particularly useful for identifying and documenting design functions. Designs are represented as an abstract hierarchy of functions. The functions at the top of the hierarchy denote the overall functional capabilities of a program or system and may correspond to requirements functions. Functions at lower levels correspond to the functional capabilities required to implement the higher level functions. General design functions often correspond to modules or parts of programs which are identified as separate functions by comments. Detailed design functions may be invented during the programming stage of system development and may correspond to single lines of code.

4.26.4. Information output. The output to be examined depends on the nature of the tested function. If it is a straight input/output function, then output values are examined. The testing of other classes of functions may involve the examination of the state of a data base or file.

4.26.5. Outline of method. The basic idea in functional testing is to identify "functionally important" classes of data. The two most important classes of data are extremal values and special values. Different kinds of sets of data have different kinds of extremal values and different classes of special values must be used to test different kinds of functions.

a. Extremal values. The simplest kinds of extremal values are associated with elementary data items. If a variable is constrained to take on values which lie in the range (a,b), then the extremal values are a and b. If a variable is constrained to take on values from a small set of discrete values then each of those values can be thought of as an extremal case.

The construction of extremal cases for data structures (e.g., group data items) can be more complicated. It is necessary to construct extremal values of both the component elementary parts of the data structure as well as its dimensions. The data structure can be treated as a single quantity. In this case, when it takes on an extremal value all of its elements take on that value. It is also possible to consider its components as a set of values in which one, more, or all of the components have extremal values. The construction of extremal values for files and data bases is similar to that for data structures. Files with extremal dimensions contain the smallest possible and largest possible number of records. If the records are variable sized they contain records of the smallest and largest dimensions.

b. Special values. There appear to be two kinds of special values that are important for data processing programs. The first is useful for testing functional capabilities in which data is moved around from one location to another, as in a transaction-update program. Functions of this type should be tested over distinct sets of data (i.e., values in different files, records, variables or data structure elements should be different) in order to detect the transfer of the incorrect data from the wrong source or into the wrong destination. The second kind of special data is useful for testing logical functional capabilities that carry out different operations on the basis of relationships between different data items. It is important to test functional capabilities of this type over special values such as those in which sets of data that enter into the comparison are all the same.

Additional kinds of special values are important for scientific programs or programs which do arithmetic calculations. They include zero, positive and negative values "close" to zero, and large negative and positive values.

Functional testing requires that tests be constructed in which the input data is extremal, non-extremal and special as well as tests that result in program output that is extremal, non-extremal or special.

4.26.6. Examples.

Example 1: Testing of requirements functions.

a. Application. A computerized dating system was built in which a sequential file of potential dates was maintained. Each client for the service offered would submit a completed questionnaire which was used to find

the five most compatible dates. Certain criteria had to be satisfied before any potential data was selected and it is possible that no date could be found for a client or less than five dates found.

b. Error. An error in the file processing logic causes the program to select the last potential date in the sequential file whenever there is no potential date for a client.

c. Error discovery. The number of dates which are found for each client is a dimension of the output data and has extremal values 0 and 5. If the "find-a- date" functional capability of the system is tested over data for a client for which no date should exist then the presence of the error will be revealed.

Example 2: Testing of detailed design functions.

a. Application. The designer of the computerized dating system in Example 1 decided to process the file of potential dates for a client by reading in the records in sets of 50 records each. A simple function was designed to compute the number of record subsets.

b. Error. The number of subsets function returns the value 2 when there are less than 50 records in the file.

c. Error discovery. The error will be discovered if the design function is tested over the extremal case for which is should generate the minimal output value 1. Note that this error is not revealed (except by chance) when the program is tested at the requirements specifications level. It will also not necessarily be revealed unless the code implementing the design function is tested independently and not in combination with the rest of the system.

4.26.7. Effectiveness. Studies have been carried out which indicate functional testing to be highly effective. Its use depends on specific descriptions of system input and output data and a complete list of all functional capabilities. The method is essentially manual and somewhat informal. If a formal language could be designed for describing all input and output data sets then a tool could be used to check the completeness of these descriptions. Automated generation of extremal, non-extremal and special cases might be difficult since no rigorous procedure has been developed for this purpose.

For many errors it is necessary to consider combinations of extremal, non-extremal and special values for "functionally related" input data variables. In order to avoid combinatorial explosions, combinations must be restricted to a small number of variables. Attempts have been made to identify important combinations (see references) but there are no absolute rules, only suggestions and guidelines.

4.26.8. Applicability. This method is generally applicable.

4.26.9. Learning. It is necessary to develop some expertise with the identification of extremal and special cases and to avoid the combinatorial explosions that may occur when combinations of extremal and special values for different data items are considered. It is also necessary to become skilled in the identification of specifications functions although this process is simplified if a systematic approach is followed for the representation of requirements and design.

4.26.10. Costs. The method requires no special tools or equipment and contains no hidden excessive tests.

4.26.11. References.

(1) HOWDEN, William E., "Functional Program Testing," IEEE Transactions on Software Engineering, SE-7, March, 1980.

(2) HOWDEN, William E., "Functional Testing and Design Abstractions," Journal of Systems and Software, Vol. 1, 307-313, 1980.

(3) MYERS, Glenford, "The Art of Software Testing," Wiley-Interscience, New York, 1975.

4.27.1. Name. Symbolic execution.

4.27.2. Basic features. Symbolic execution is applied to paths through programs. It can be used to generate expressions which describe the cumulative effect of the computations which occur in a program path. It can also be used to generate a system of predicates describing the subset of the input domain which causes a specified path to be traversed. The user is expected to verify the correctness of the output which is generated by symbolic execution in the same way that output is verified which has been generated by executing a program over actual values. It is used as a basis for data flow analysis and proof of correctness.

4.27.3. Information input.

a. Source code. The method requires the availability of the program source code.

b. Program paths. The path or paths through the program which are to be symbolically evaluated must be specified. The paths may be specified directly by the user or, in some symbolic evaluation systems, selected automatically.

c. Input values. Symbolic values must be assigned to each of the "input" variables for the path or paths which are to be symbolically evaluated. The user may be responsible for selecting these values or the symbolic evaluation system which is used may select them automatically.

4.27.4. Information output.

a. Values of variables. The variables whose final symbolic values are of interest must be specified. Symbolic execution will result in the generation of expressions which describe the values of these variables in terms of the dummy symbolic values assigned to input variables.

b. System of predicates. Each of the branch-predicates which occur along a program path constrains the input which causes that path to be followed. The symbolically evaluated system of predicates for a path describes the subset of the input domain that causes that path to be followed.

4.27.5. Outline of method.

a. Symbolic execution. Symbolic values are symbols standing for sets of values rather than actual values. The symbolic execution of a path is carried out by symbolically executing the sequence of assignment statements occurring in the path. Assignment statements are symbolically executed by symbolically evaluating the expressions on the right hand side of the assignment. The resulting symbolic value becomes the new symbolic value of the variable on the left hand side. An arithmetic or logical expression is symbolically executed by substituting the symbolic values of the variables in the expression for the variables.

The branch conditions or branch predicates which occur in conditional branching statements can be symbolically executed to form symbolic predicates. The symbolic system of predicates for a path can be constructed by symbolically executing both assignment statements and branch predicates during the symbolic execution of the path. The symbolic system of predicates consists of the sequences of symbolic predicates that are generated by the execution of the branch predicates.

b. Symbolic execution systems. All symbolic execution systems must contain facilities for: selecting program paths to be symbolically executed, symbolically executing paths, and generating the required symbolic output.

Three types of path selection techniques have been used: interactive, static and automatic. In the interactive approach, the symbolic execution system is constructed so that control returns to the user each time it is necessary to make a decision as to which branch to take during the symbolic execution of a program. In the static approach, the user specifies the paths he wants executed in advance. In the automatic approach, the symbolic execution system attempts to execute all those program paths having consistent symbolic system of predicates. A system of predicates is consistent if it has a solution.

The details of symbolic execution algorithms in different systems are largely technical. Symbolic execution systems may differ in other than technical details in the types of symbolic output they generate. Some systems contain, for example, facilities for solving systems of branch predicates. Such systems are capable of automatically generating test data for selected program paths (i.e., program input data which will cause the path to be followed when the program is executed over that data).

4.27.6. Example.

a. Application. A FORTRAN program called SIN was written to compute the sine function using the McLaurin series.

```
 PREDICATES:

  (X**3/6).GE.E
  (X**5/120).GE.E
  (X**7/5040).LT.E
 OUTPUT

  SIN = ?SUM  - (X**3/6) - (X**5/120)
    Symbolic output for SIN
```

Figure 4.27.6-1 Symbolic Execution Example

b. Errors. The program contained three errors, including an uninitialized variable, the use of the expression -1**(I/2) instead of (-1)**(I/2), and the failure to add the last term computed in the series on to the final computed sum.

Different paths through SIN correspond to different numbers of iterations of the loop in the program that is used to compute terms in the series. The symbolic output in figure 4.27.6-1 was generated by symbolically evaluating the path that involves exactly three iterations of the loop.

c. Error discovery. The errors in the program are discovered by comparing the symbolic output with the standard formula for the McLaurin series. The symbolic evaluator that was used to generate the output represents the values of variables that have been uninitialized with a question mark and the name of the variable. The error involving the expression (-1)**(I/2) results in the generation of the same rather than alternating signs in the series sum. The failure to use the last computed term can be detected by comparing the predicates for the symbolically evaluated path with the symbolic output value for SIN.

4.27.7. Effectiveness. Studies have been carried out which indicate that symbolic evaluation is useful for discovering a variety of errors but that, except in a small number of cases, it is not more effective than the combined use of other methods such as dynamic and static analysis (1).

One of the primary uses of symbolic evaluation is in raising the confidence level of a user in a program. Correct symbolic output expressions confirm to the user that the code carries out the desired computations. It is especially useful for nonprogrammer users.

4.27.8. Applicability. The method is primarily useful for programs written in languages which involve operations that can be represented in a concise formal way. Most of the symbolic evaluation systems that have been built are for use with algebraic programming languages such as FORTRAN and PL-1. Algebraic programs involve computations that can be easily represented using arithmetic expressions. It is difficult to generate symbolic output from programs which involve complex operations with "wordy" representations such as the REPLACE and MOVE CORRESPONDING operations in COBOL.

4.27.9. Learning. It takes a certain amount of practice to choose paths and parts of paths for symbolic evaluation. The user must avoid the selection of long paths or parts of paths that result in the generation of expressions that are so large that they are unreadable. If the symbolic evaluation system being used gives the user control over the types of expression simplification that are carried out, then he must learn to use this in a way that results in the generation of the most revealing expressions.

4.27.10. Costs. Storage and execution time costs for symbolic evaluation have been calculated in terms of program size, path length, number of program variables and the cost of interpreting (rather than compiling and executing) a program path.

The storage required for symbolically evaluating a path of length P in a program with S statements containing N variables is estimated to be on the order of 10(P+S+V) (2). Let C1 be the cost of preprocessing a program for interpretation, C2 the cost of interpreting a program path, Cons is the cost of checking the consistency (i.e., solvability) of a system of symbolic

predicates and Cond is the cost of evaluating a condition in a conditional statement. Cons and Cond are expressed in units of the cost of interpreting a statement in a program. The cost (in execution time) of symbolically executing a program path is estimated to be on the order of C1 + C2 (1 + E + Cons/10 + Cond/100) (2).

4.27.11. References.

(1) HOWDEN, William E., "An Evaluation of the Effectiveness of Symbolic Testing," Software--Practice and Experience,8, 1978.

(2) HOWDEN, William E., "Symbolic Testing -- Design Techniques, Costs and Effectiveness," U.S. Department of Commerce,NTIS PB-268,517, Springfield, Virginia.

(3) HOWDEN, William E., "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Transactions on Software Engineering, SE-3, 1977.

(4) KING, J.C., "Symbolic Execution and Program Testing," CACM, 19,1976.

(5) CLARKE, L.A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering,SE-2, 1976.

4.28.1. Name. Test coverage analyzers.

4.28.2. Basic features. Test coverage analyzers monitor the execution of a program during program testing in order to measure the completeness of a set of program tests. Completeness is measured in terms of the branches, statements or other elementary program constructs which are used during the execution of the program over the tests.

4.28.3. Information input. Test coverage analyzers use the program source code and a set of program tests to generate test coverage reports. Sophisticated coverage analyzers may also involve input parameters that describe which of several alternative coverage measures are to be used.

4.28.4. Information output. Typical output consists of a report which describes the relevant feature of the program which has been "exercised" over a sequence of tests. Branch coverage analyzers keep track of and report on the number of times that each branch in a program has been traversed during a sequence of tests (1). A program branch is any transfer of control from one program statement to another, either through execution of a control transfer instruction or through normal sequential flow of control from one statement to the next.

Different kinds of coverage analyzers will report different kinds of information. Analyzers which measure coverage in terms of pairs of branches, loop iteration patterns or elementary program functions have been proposed but branch coverage analyzers are the most widely used. In addition to coverage information, analyzers may also record and print variable range and subroutine call information. The minimum and maximum values assumed by each variable in a program, the minimum and maximum number of times that loops are iterated during the executions of a loop, and a record of each subroutine call may be reported.

4.28.5. Outline of method.

a. Branch analyzers. Branch coverage analyzers typically consist of two parts, a preprocessor and a postprocessor. The preprocessor inserts "probes" into the program for which test coverage analysis is required.

The probes call subroutines or update matrices that record the execution of the part of the program containing the probe. Theoretical studies have been carried out to determine the minimum number of probes required to determine which branches are executed during a program execution. The probes may also record information for determining minimal and maximal variable values, loop iteration counts and subroutine calls.

The information which is generated by program probes has to be processed before test coverage reports can be generated. If a sequence of tests has been carried out, the information from the different tests has to be merged. The processing of the information generated by probes during program testing is processed and reports are generated by the coverage analyzer postprocessor.

b. Function analyzers. Function analyzers are based on the idea that each program construct implements one or more elementary functions. Loop constructs, for example, involve functions which determine if a loop is to be entered, when it is to be exited, how many times it is to be iterated, the initial value of the loop index variable (if present) and subsequent values of the loop index. It is possible to define complete sets of tests for these functions which will cause the function to act incorrectly on at least one test if the function contains one of a predefined set of possible functional errors (2). Test coverage analyzers can be built which keep track of the data over which constructs are executed and which report on the functional completeness of the data used in the execution of the constructs. Function coverage analyzers can be constructed using the preprocessor probe insertion and postprocessor report generation approach used for branch coverage analyzers.

4.28.6. Example.

a. Application. A quicksort program was constructed which contains a branch to a separate part of the program code that carries out an insertion sort. The quicksort part of the code branches to the insertion sort. The quicksort part of the code branches to the insertion sort whenever the size of the original list to be sorted or or a section of the original list is below some threshold value. Insertion sorts are more effective than quicksorts for small lists and sections of lists because of the smaller constants in their execution time formulae.

b. Error. The correct threshold value is 11. Due to a typographical error, the branch to the insertion sort is made whenever the length of the original list, or the section of the list currently being processed, is less than or equal to one.

c. Error discovery. Parts of the insertion sort code are not executed unless the list or list section being sorted is of length greater than one. Examination of the output from a branch coverage analyzer will reveal that parts of the program are never executed, regardless of the program tests which are used. This will alert and draw the attention of the programmer to the presence of the error.

It is interesting to note that ts error is not discoverable by the examination of test output data alone since the program will still correctly sort lists.

4.28.7. Effectiveness. Research results confirm that test coverage analyzers are a necessary and important tool for software validation. Previously assumed "complete" test sets for production software have been found to test less than 50% of the branches in a program (1). The use of test coverage analyzers reveals the inadequacy of such test sets.

Studies indicate that although test coverage of all parts of a program is important, it is not enough to simply test all branches, or even all program paths. A large percentage of errors are only detectable when a program is tested over extremal cases or special values that are closely related to the functions performed in the program. There appear to be three situations in

which branch coverage is effective in finding errors. The first is that in which an error in part of a program is so destructive that any test that causes that part of the program to be executed will result in incorrect output. The second is that in which parts of a program are never used during any program execution, and the third that in which unexpected parts of a program are used during some test. Other kinds of errors require additional test selection techniques, such as functional testing.

4.28.8. Applicability. Test coverage analysis can be applied to any kind of program in any programming language.

4.28.9. Learning. There are no special learning requirements for the use of test coverage analyzers. Once a set of tests has been found to be inadequate, it requires skill to generate data that will cause the unexercised features of the program to be used during program execution.

4.28.10. Costs. Test coverage analyzers can be inexpensive to use. The major expense is the capital cost for the tool. It is estimated that the construction of a test coverage tool requires a level of effort which is more than that required for a parser but less than twice that effort. The major part of test coverage analyzers consists of the parser that is used to determine probe insertion points for a program.

4.28.11. References.

(1) STUCKI, Leon G., "Automatic Generation of Self-metric Software," Proc. 1973 IEEE Symposium on Computer Software Reliability ,94 (1973).

(2) HOWDEN, William E., "Completeness Criteria for Testing Elementary Program Functions," University of Victoria, Dept. of Mathematics, DM-212-IR, May 1980.

(3) GANNON,Carolyn,"Error Detection Using Path Testing and Static Analysis",Computer,August 1979.

4.29.1. Name. Test data generators.

4.29.2. Basic features. Test data generators are tools which generate test data to exercise a target program. They may generate data through analysis of the program itself or through analysis of the expected input to the program in its normal operating environment. Test data generators may use numerical integrators and random number generators to create the data.

4.29.3. Information input. Test data generators require as input:

a. the program for which data is to be generated, or
b. a quantifiable description of the domain of possible inputs to the program from which the test data generator is to produce representative values.

4.29.4. Information output. The output produced by test data generators is a set of data that can be used effectively to detect execution-time errors in a program. It is generally intended that such test data cause the program to be thoroughly exercised when executed. It is also desirable to have this input data be representative of the actual data used in real program operation in order to properly evaluate results obtained from program execution.

4.29.5. Outline of method. Test data generators generate test data for a program in a systematic, deterministic manner. There are two major methods currently used to generate test data. Both methods can be implemented as fully automated tools.

One method of test data generation analyzes the structure of a program and, based upon this analysis, generates a set of test data which will drive execution along a comprehensive set of program paths. This method attempts to maximize the structural coverage achieved during execution with the derived data. Though this approach requires a detailed, rigorous structural analysis of a program (which is often quite difficult, if not impossible), tools have been developed which aid in the automation of this analysis. There are tools which can analyze a program and identify certain structural elements in that program. Data is then automatically generated that will drive execution through each of these program elements.

If it is desirable to increase the coverage achieved by the test data, there also exist tools which use automated program analysis to aid in accomplishing this. After monitoring program execution with the generated data, it may be possible to increase the current structural coverage achieved by using automated tools which assist in determining how to alter the current set of test data as necessary to cause different branching conditions to occur. Test data generators that create test data based upon the amount of structural coverage that the data will achieve are generally very sophisticated tools. Much research and development work is currently being done in this area.

A second approach to generating test data is based upon analysis of the possible inputs to a program under real, operational usage. This technique requires more knowledge of the software for which input data is to be generated than the previous technique. However, in this approach the output generated from program execution provides more meaningful results to the user

during testing. One such tool that utilizes this technique examines the domain of all possible input values to a program under normal program operation and partitions this domain into mutually exclusive subdomains. For each subdomain there is an associated probability that a sequence of actual input values will belong to that partition. Data is then generated by sampling from each subdomain with the distribution of sampling determined by the subdomain's associated probability. Automated tools have been built to assist in computing these probabilities and in sampling from the appropriate partitions.

This technique attempts to mirror the intended operation of a program by generating test data which is representative of its operational input. This mode of program testing can be very useful during a preliminary period of software operational use. Using this technique, reasonably accurate predictions can be made on the software's performance in real operation.

Other test data generators exist which use less sophisticated techniques than those described above. Many of them generate data based upon commands given by the user and/or from data descriptions in a program, such as in a COBOL program's data definition section. This is mainly a COBOL oriented technique in which the test data is intended to simulate transaction inputs in a database management situation. This technique, however, can be adapted to other environments.

4.29.6. Example. Test data is required for a new payroll program. A test data generator is used to generate data normally contained in the payroll records of each employee on the payroll. The data fields in these records consists of:

- o Employee identification number
- o Employee name
- o Indication of hourly or salaried employee
- o Salary rate (if salaried)
- o Hourly rate (if hourly)
- o Number of hours worked during last pay period
- o Number of tax exemptions declared
- o Federal withholding tax rate
- o Social security tax rate
- o Marital status

A file of records containing this information is created by the test data generator. For each field in a record, a value with the appropriate data type is randomly generated (e.g., alphanumeric for Employee Name, integer for Employee Identification Number, real for Federal Withholding Tax Rate). The file is then reformatted in an organization that is acceptable to the payroll system as input. The generated test data will then be fed to the payroll program to be tested.

4.29.7. Effectiveness. The overall effectiveness of automated test data generators in use today is generally poor. Though these tools permit the generation of more test data than any human tester could create (thereby devising more test cases), a burden is created on the human tester to evaluate

all the test results obtained from program execution with the generated data. Unfortunately, test data generators themselves do not have a facility by which to verify these test results. In addition, most of the test data generators in use today create data in a manner which is totally insensitive to the functional peculiarities of a program. The data may often be meaningless in content. It may focus testing upon an unimportant portion of the program and totally ignore critical portions. A human tester, however, often has a certain intuition about which program areas need to be more thoroughly tested than others and so creates his test data accordingly. The overall ignorance of test data generators in determining which data items would offer the most potential in discovering errors is the major factor behind their current ineffectiveness in program testing.

4.29.8. Applicability. Test data generators are generally applicable for any system requiring input data for operation.

4.29.9. Learning. For those test data generators which only require as input the source program for which test data is desired, very little learning is required to use these tools. The user interface with the tool will always be the same, and the User Manual for the tool should provide sufficient information for its operation. For those data generators which create data based upon the domain of expected inputs to the program, much more learning is required. It is necessary to acquire some knowledge about the application environment and operational usage of the software so that representative input data can be generated.

4.29.10. Costs. Automated test data generators are generally quite expensive. This is primarily due to the relatively infrequent use of these tools in actual testing environments. The initial costs in building test data generators have very rarely been offset by benefits obtained in using them. As yet, the derived utilization of the more sophisticated tools that exist have not justified their cost. Accordingly, test data generators are among the most costly testing tools that exist today.

4.29.11. References.

(1) CLARKE, L.A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, SE-2, September, 1976.

(2) HOWDEN, W.E., "Methodology for Generation of Program Test Data," IEEE Transactions on Computers, TC-24, May, 1975.

(3) MILLER, E.F. and MELTON, R.A., "Automated Generation of Testcase Datasets," 1975 International Conference on Reliability, Los Angeles, April, 1975.

(4) NAFTALY, S.M. and COHEN, M.C., "Test Data Generators and Debugging Systems . . .", Workable Quality Control, Part I and II, Data Processing Digest, Vol 18, 2 and 3, February and March, 1972.

4.30.1. Name. Test support facilities.

4.30.2. Basic features. An environment simulation, or test bed, is a test site used to test a component of software. This test site simulates the environment under which the software will normally operate. A test bed permits full control of inputs and computer characteristics, allows processing of intermediate outputs without destroying simulated execution time, and allows full test repeatability and diagnostics. To be effective, the controlled circumstances of the test bed must truly represent the behavior of the system of which the software is a part.

4.30.3. Information input. The information input to a test bed is the software for which a testing environment is to be simulated and which will later be installed in a real system.

4.30.4. Information output. The information output by a test bed are the results observed through execution of the software installed in the test bed. This information is used as a preliminary means of determining whether the software will operate as intended in its real environment.

4.30.5. Outline of method. Test beds provide an environment in which to monitor the operation of software prior to installation in a real system. To be of value, this environment must realistically reflect those properties of the system which will affect or be affected by the operation of the software. However, the test bed should simulate only those components in the system which the software requires as a minimum interface with the system. This will permit testing to focus only on the software component for which the test bed is built.

Test beds are built through the consideration of, and proper balance between, three major factors:

- o the amount of realism required by the test bed to properly reflect the operation of system properties,
- o resources available to build the test bed, and
- o the ability of the test bed to focus only on the software being tested.

Test beds come in many forms, depending on the level of testing desired. For single module testing, a test bed may consist merely of test data and a test driver. A test driver is a program which feeds input data to the program module being tested, causes the module to be executed, and collects the output generated during the program execution. If a completed, but non-final version of software is to be tested, the test bed may also include stubs. A stub is a dummy routine that simulates the operation of a module that is invoked within a test. Stubs can be as simple as routines that automatically return on a call, or they can be more complicated and return simulated results. The final version of the software may be linked with other software subsystems in a larger total system. The test bed for one component in the system may consist of those system components which directly interface with the component being tested.

As illustrated in the above examples, test beds permit the testing of a component of a system without requiring the availability of the full, complete system. They merely supply the inputs required by the software component to be executed and provide a repository for outputs to be placed for analysis. In addition, test beds may contain monitoring devices which collect and display intermediate outputs during program execution. In this way, test beds provide the means of observing the operation of software as a component of a system without requiring the availability of other system components, which may be unreliable.

4.30.6. Example. The federal government has just distributed to all American corporations new tax rates to be imposed on the earnings of all employees beginning at the start of next year. Due to these new tax rates, Company XYZ has had to revise its current payroll program so that it will accommodate the new federal regulations by January 1.

In order to test this new program, a test bed is being constructed to simulate the operation of the payroll system. To simulate the inputs to this system, a test file of data containing all the information necessary for the system to operate is created. The file consists of a record of information for each employee in the company. Each record contains the following data:

- o Employee identification number
- o Employee name
- o Indication of hourly or salaried employee
- o Salary rate (if salaried)
- o Hourly rate (if hourly)
- o Number of hours worked during last pay period
- o Number of tax exemptions declared
- o Federal withholding tax rate
- o Social security tax rate
- o Marital status

A test driver controls the execution of the payroll program. It feeds the above data to the program in the proper format. At the end of program execution, the driver simulates the check-writing facility of the payroll system in the following manner. It directs the output of the payroll program to an output file. The output consists of a record of data for each company employee. Each record contains the following information:

- o Employee name
- o Employee social security number
- o Check date
- o Total employee earnings less deductions

The test driver then dumps this information from the output file onto a hardcopy device so that the output can be analyzed and verified for correctness.

4.30.7. Effectiveness. The use of test beds has proven to be a highly effective and widely used technique to test the operation of software. The use of test drivers, in particular, is one of the most widely used testing

techniques.

4.30.8. Applicability. This method is generally applicable, from single module to large system testing and for all types of computing applications.

4.30.9. Learning. In order to build an effective test bed, it is necessary to develop a solid understanding of the software and its dynamic operation in a system. This understanding should aid in determining what parts of the test bed deserve the most attention during its construction. In addition, knowledge of the dynamic nature of a program in a system is required in gathering useful intermediate outputs during program execution and in properly examining these results.

4.30.10. Cost. The amount of realism desired in a test bed will be the largest factor affecting cost. Building a realistic test bed may require the purchasing of new hardware and the development of additional software in order to properly simulate an entire system. In addition, these added resources may be so specialized that they may seldom, if ever, be used again in other applications. In this way, very sophisticated test beds may not prove to be highly cost-effective.

4.30.11. References.

(1) HARTWICK, R.D., "The Advanced Targeting Study," SAMSO-TR-71-124, Volume 1, June 1971.

(2) PANZL, D.J., "Automatic Software Test Drivers," IEEE Computer, April 1978.

4.31.1. Name. Walkthroughs.

4.31.2. Basic features. Walkthroughs (WT) constitute a structured series of peer reviews of a system component used to enforce standards, detect errors, and improve development visibility and system quality. They may be conducted during any of the lifecycle phases and may also be applied to documentation. An identifying feature of a WT is that it is generally presented by the creator or producer of the material being reviewed rather than an independent or third party. In addition, because of the presenter's advance preparation and his familiarity with the material, less preparation by other members is required.

4.31.3. Information input.

a. Walkthrough Package. This set of materials includes all necessary backup documentation for the WT. Examples of materials made available include (but are not limited to) module flow charts, system flow charts, HIPO charts (or other high-level representation schemes), and module listings. Other important materials may include sections of the Functional Specification, System/Subsystem Specification and Database Specification (as applicable) which pertain to the component under review. Often, copies of applicable standards are also part of the WT input.

b. Questions List. Some organizations which practice a more formal version of a WT require reviewers to submit the component to the presenter prior to the WT. This enables the presentor to be better prepared to respond to the questions at the WT.

4.31.4. Information output.

a. Action List. During the WT, a list of problems and questions is recorded. This action list is distributed to all participants and is used by the producer (reviewee) as the basis for subsequent changes to the component.

b. Walkthroughs Form. During the course of the WT, this form is completed by an individual with recording responsibilities. The form identifies participants and their responsibilities, the agenda for the WT, the decision of the WT (accept as-is, revise, revise and schedule another WT), and is signed by all participants at the end of the WT.

4.31.5. Outline of method.

a. Roles and Responsibilities. The group of individuals participating in a WT are usually referred to as reviewers. The leader of the WT is called the coordinator. The coordinator is responsible for WT planning, organization, and distribution of materials. The WT is called to order, moderated, and summarized by the coordinator.

The producer (or reviewee) is that individual whose module or component is to be reviewed during the WT. In most cases, the producer is generally responsible for selecting the coordinator and review team (in most situations; sometimes management may perform this function) and providing the WT package

materials to the coordinator. During the WT the producer initially provides a general description of the module, then leads the reviewers through a detailed, step-by-step description of the module. After the WT the producer should objectively consider every item on the action list and make changes to his product as he deems appropriate.

The reviewers are composed of individuals from varying backgrounds and fulfill responsibilities based upon their area of specialization. Some roles which are fulfilled are those of recorder and representatives of the user, standards and maintenance groups. In general, these participants are responsible for being familiar with the material being presented, submitting comments prior to the review, and listening and contributing during the WT. At the end of the review each must cast a vote indicating whether the module is acceptable, needs revision, or is rejected.

Because of the organization which each is representing, some specific responsibilities are associated with each reviewer. In addition to contributing to the WT, the recorder must make written note of the participants assembled and the action items which result from the review.

The user representative is often involved during early WT's of a module (i.e., during requirements analysis and design). His responsibility is to ensure that the proposed solution is usable and does, in fact, meet the needs of his organization.

The standards representative, referred to by some sources as a "standards bearer," is responsible for checking that the product being reviewed adheres to organization standards. In some cases, he may be asked to provide input to a request to deviate from a standard.

The maintenance representative, referred to by some sources as the "maintenance oracle," must view the product from the standpoint of the group who will be required to maintain the product. Items which may be of prime concern to this individual are documentation and program comments, program functionality or modularity, naming conventions, and data decomposition.

b. The Process. Many organizations practice walk-throughs which differ radically in formality. The process described in the following paragraphs falls at the midpoint between these extremes. There are four basic steps in the process.

. Scheduling. When the work item module is very near completion (including documentation), the producer notifies management and selects the WT participants. The WT date is agreed upon and facilities are scheduled. The WT should not exceed 2 hours and is best is kept to less than 1 hour. This implies that the work item is of manageable size. Sources suggest the following guidelines for work package size:

- o 5-10 pages of specifications for a requirements WT,
- o 1-5 structure charts (or HIPO diagrams) for a preliminary or detailed design WT,
- o 50-100 lines of code for a code or test WT.

2. Preparation. The producer collects appropriate information for use at the WT and gives it to the coordinator for distribution. Each reviewer studies the materials, making a note of questions or comments. Most sources estimate that a maximum of 1 hour preparation by reviewers is necessary.

3. Walkthrough Meeting. After the coordinator opens the review, the producer uses test data to simulate the operation of the component. Each specification, design phrase, or line of code is reviewed. The recorder documents comments or questions usng the action list. Each reviewer signs the Walkthrough, documenting the decision of the meeting (accept product as-is, accept with modification, or reject). The recorder provides a copy of action list to all participants and supplies a copy of the Walkthrough Form to management.

4. Re-Work. The producer reviews each action item, making product changes as he feels necessary. He may decide to implement all, part or none of the suggested changes. No follow-up is held to ensure that suggestions are incorporated; it is assumed that the producer is in the best position to make implementation decisions. Major items on the action list may be summarized at the next WT for the module.

4.31.6. Example. One week prior to completion of coding of a module of 75-100 lines, the producer notifies his line manager of the need for a WT. Upon management approval the producer selects a coordinator (one of the lead analysts from the development shop), a standards representative (from the Quality Assurance group), a maintenance representative (from the Production Program organization), and a user representative (from the group requesting the system). Three days prior to the inspection he notifies the coordinator of the planned WT and suggested participants. At this time he gives the coordinator copies of the program listing (including comments), a systems-level flowchart depicting how it interfaces with other modules, a data dictionary, a set of test data items, and a section from the Functional Specification detailing the user requirement associated with the module.

The coordinator notifies the selected participants, receives their commitment to attend and distributes to each a copy of the materials furnished by the producer.

Each participant reviews the materials. The standards representative finds two instances of deviations from published standards and notifies the coordinator (who in turn notifies the producer). The user representative verifies that the code addresses each designed aspect by reviewing the proceedings of the previous design WT. He is satisfied that each requirement has been addressed and notifies the coordinator that he finds no errors and feels that his presence is not required for the code walkthrough. The maintenance representative finds no immediate concerns with the code but makes a note to inquire about the structure of the data files.

The WT begins with a brief introduction by the coordinator, who then turns the review over to the producer. He uses the system flowchart to give a summary of the functions of the module and proceeds to go line-by-line through the

code using the selected test data. Upon reaching the lines of concern to the standards representative, a brief discussion occurs to explain the reasons for the deviations from standard. In this instance, the reviewers are satisfied that the deviations are justified. The recorder so notes on the action list and the meeting proceeds. The maintenance representative points out one line of highly complex code and suggests that it be broken up into two less complex steps. Agreement cannot be immediately reached, so the suggestion is added to the action list.

At the end of the module review the coordinator seeks a decision from the reviewers about the module. They agree to give their approval, providing that the suggested changes are made and that the producer will further investigate the effect of breaking up the complex line of code. Each signs the Walkthrough form and the meeting is adjourned.

The recorder distributes a copy of the action to all participants. The producer makes the changes he feels are necessary. He runs a benchmark of the module with the complex code and again with the code broken down. Since no significant loss of efficiency resulted, he modifies the code. The module is now ready for unit test which may be followed by another WT.

4.31.7. Effectiveness. Studies have been conducted which identify the following qualitative benefits of Walkthroughs:

- o higher status visibility
- o decreased debugging time
- o early detection of design and analysis errors which would be much more costly to correct in later develoment phases
- o identification of design or code inefficiencies
- o ensuring adherence to standards
- o increased program readability
- o increased user satisfaction
- o communication of new ideas or technology
- o increased maintainability

Little data is available which identifies the quantitative benefits attributable to the use of Walkthroughs. However, one source estimates that the number of errors in production programs was reduced by a factor of ten.

4.31.8. Applicability. The Walkthrough is applicable to large or small projects during all development phases and is not limited by project type or complexity.

4.31.9. Learning. The Walkthrough does not require special training to implement. However, experience has shown that the effectiveness of the Walkthrough increases as the WT experience of the reviewers increases.

4.31.10. Costs. The WT requires no special tools or equipment to implement. The direct costs are equal to the expense associated with the human resources involved.

4.31.11. References.

(1) "Code Reading: Structured Walkthroughs and Inspections", IBM IPTO Support Group, World Trade System Center, Postbus 60, Zoetermeer, Netherlands, March 1976.

(2) FAGAN, M. E., "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, No. 3, 1976.

(3) FREEDMAN, D. P., and WEINBERG, G. M., "Ethno - Technical Review Handbook", Ethnotech, Inc., 1977.

(4) DALY, E. B., "Management of Software Development", IEEE Transactions on Software Engineering, May 1977.

(5) SHNEIDERMAN, Ben, "Software Psychology - Human Factors in Computer and Information Systems," Winthrop Publishing, 1980.

GLOSSARY

BLACK BOX TESTING: see FUNCTIONAL TESTING

BOUNDARY VALUE ANALYSIS: a selection technique in which test data is chosen to lie along "boundaries" or extremes of input domain (or output range) classes, data structures, procedure parameters, etc. Choices often include maximum, minimum, and trival values or parameters. This technique is often called stress testing.

BRANCH TESTING: a test method satisfying coverage criteria that require, for each decision point, each possible branch be executed at least once.

CAUSE-EFFECT GRAPHING: test data selection technique. The inputs and outputs of the program are determined through analysis of the requirements. A minimal set of inputs is chosen avoiding the testing of multiple inputs which cause identical output.

COMPLETENESS: the property that all necessary parts of the entity in question are included. Completeness of a product is often used to express the fact that all requirements have been met by the product.

CONSISTENCY: the property of logical coherency amoung constitutant parts. Consistency may also be expressed as adherence to a given set of rules.

CORRECTNESS: the extent to which software is free from design and coding defects, i.e. fault free. It is also the extent to which software meets its specified requirements and user objectives. (IEEE Software Engineering Terminology)

DEBUGGING: the process of correcting syntactic and logical errors detected during coding. With the primary goal of obtaining an executing piece of code, debugging shares with testing certain techniques and strategies but differs in its usual ad hoc application and local scope.

DESIGN BASED FUNCTIONAL TESTING: the application of test data derived through functional analysis (see FUNCTIONAL TESTING) extended to include design functions as well as requirement functions.

DRIVER: code which sets up an environment and calls a module for test.

DYNAMIC ANALYSIS: involves execution or simulation of a development phase product. It detects errors by analyzing the response of a product to sets of input data.

EXTREMAL TEST DATA: test data that is at the extremes, or boundaries, of the domain of an input variable or which produces results at the boundaries of an output domain.

FORMAL ANALYSIS: uses rigorous mathematical techniques to analyze the algorithms of a solution. The algorithms may be analyzed for numerical properties, efficiency, and/or correctness.

FUNCTIONAL TESTING: application of test data derived from the specified functional requirements without regard to the final program structure.

INSPECTION: a manual analysis technique in which the program (requirements, design, or code) is examined in a very formal and disciplined manner to discover errors.

INSTRUMENTATION: the insertion of additional code into the program in order to collect information about program behavior during program execution.

INVALID INPUT (TEST DATA FOR INVALID INPUT DOMAIN): test data that lies outside the domain of the program's function.

PATH TESTING: a test method satisfying coverage criteria that each logical path through the program be tested. Often paths through the program are grouped into a finite set of classes; one path from each class is then tested.

PROOF OF CORRECTNESS: the use of techniques of mathematical logic to infer that a relation between program variables assumed true at program entry implies that another relation between program variables holds at program exit.

REGRESSION TESTING: testing of a previously validated program which has been modified for extension or correction.

SIMULATION: use of an executable model to represent the behavior of an object. During testing the computational hardware, the external environment, and even code segments may be simulated.

SPECIAL TEST DATA: test data based on input values that are likely to require special handling by the program.

STATEMENT TESTING: a test method satisfying the criterion that each statement in a program be executed at least once during program testing.

STATIC ANALYSIS: direct analysis of the form and structure of a product without executing the product. It may be applied to the requirements, design or code.

STRESS TESTING: see BOUNDARY VALUE ANALYSIS.

STUB: special code segments that when invoked by a code segment under test will simulate the behavior of designed and specified modules not yet constructed.

SYMBOLIC EXECUTION: an analysis technique that derives a symbolic expression for each program path.

TEST DATA SET : set of input elements used in the testing process.

TEST DRIVER: a program which directs the execution of another program against a collection of test data sets. Usually, the test driver records and organizes the output generated as the tests are run.

TEST HARNESS: see TEST DRIVER.

TESTING: examination of the behavior of a program by executing the program on sample data sets.

VALID INPUT (TEST DATA FOR A VALID INPUT DOMAIN): test data that lies within the domain of the function represented by the program.

VALIDATION: determination of the correctness of the final program or software produced from a development project with respect to the user needs and requirements.

VERIFICATION: in general, the demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development lifecycle.

WALKTHROUGH: a manual analysis technique in which the module author describes the module's structure and logic to an audience of colleagues.

NOTE: Most of the definitions above are from:

ADRION,W.R.,BRANSTAD,M.A.,and CHERNIAVSKY,J.C.,"Validation, Verification, and Testing",NBS Special Publicaiton 500-75.

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NBS SP 500-93	2. Performing Organ. Report No.	3. Publication Date September 1982

4. TITLE AND SUBTITLE

Computer Science and Technology:

Software Validation, Verification, and Testing Technique and Tool Reference Guide

5. AUTHOR(S)

Patricia B. Powell, Editor

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

Boeing Computer Services Co.
Seattle, WA 98124

7. Contract/Grant No.

NB79SBCA0102

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

Same

10. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 82-600589

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

Thirty techniques and tools for validation, verification, and testing (V,V&T) are described. Each description includes the basic features of the technique or tool, the input, the output, an example, an assessment of the effectiveness and usability, applicability, an estimate of the learning time and training, an estimate of needed resources, and references.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

automated software tools; dynamic analysis; formal analysis; software testing; software verification; static analysis; test coverage; validation; V,V&T techniques; V,V&T tools.

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☒ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☐ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

138

15. Price

$6.00

USCOMM-DC 6043-P80

☆U.S. GOVERNMENT PRINTING OFFICE: 1982-360-997/2244